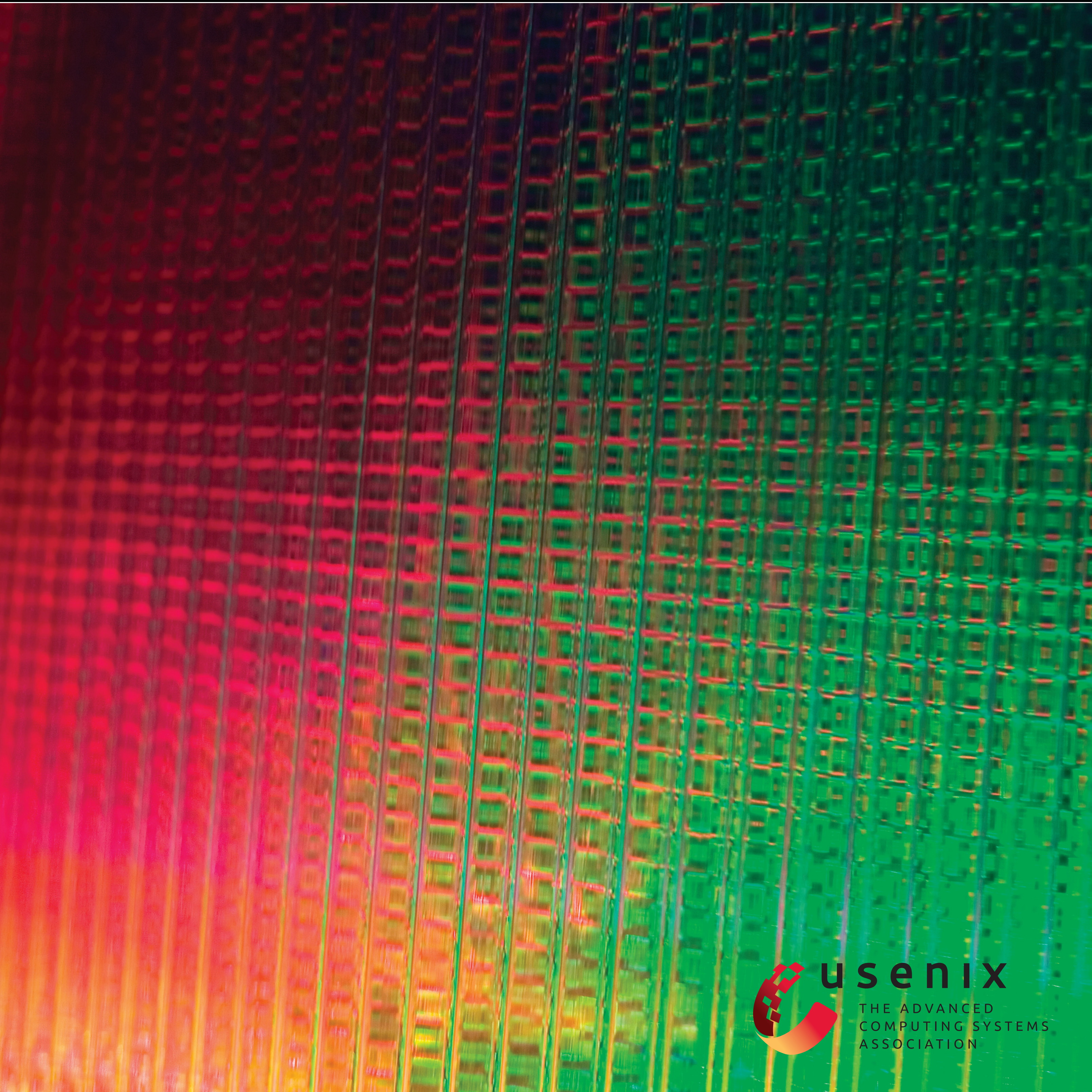
usenix ;login:

DECEMBER 2012 VOL. 37, NO. 6



usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

Middleware 2012: ACM/IFIP/USENIX 13th International Conference on Middleware
December 3–7, 2012, Montreal, Quebec, Canada
www.usenix.org/conference/middleware2012

LISA '12: 26th Large Installation System Administration Conference
December 9–14, 2012, San Diego, CA, USA
www.usenix.org/conference/lisa12

FAST '13: 11th USENIX Conference on File and Storage Technologies
February 12–15, 2013, San Jose, CA, USA
www.usenix.org/conference/fast13

TaPP '13: 5th USENIX Workshop on the Theory and Practice of Provenance
April 2-3, 2013, Lombard, IL, USA
www.usenix.org/conference/tapp13
Submissions due: January 10, 2013

NSDI '13: 10th USENIX Symposium on Networked Systems Design and Implementation
April 3–5, 2013, Lombard, IL, USA
www.usenix.org/conference/nsdi13

HotOS XIV: 14th Workshop on Hot Topics in Operating Systems
May 13–15, 2013, Santa Ana Pueblo, NM, USA
www.usenix.org/conference/hotos13
Submissions due: January 10, 2013

2013 USENIX Federated Conferences Week
June 24–28, 2013, San Jose, CA, USA

USENIX ATC '13: 2013 USENIX Annual Technical Conference
June 26–28, 2013
www.usenix.org/conference/atc13
Paper titles and abstracts due: January 23, 2013

ICAC '13: 10th International Conference on Autonomic Computing
June 26–28, 2013
www.usenix.org/conference/icac13
Submissions due: March 4, 2013

HotPar '13: 5th Workshop on Hot Topics in Parallelism
June 24–25, 2013
www.usenix.org/conference/hotpar13
Submissions due: March 7, 2013

HotCloud '13: 5th USENIX Workshop on Hot Topics in Cloud Computing
June 25–26, 2013
www.usenix.org/conference/hotcloud13
Submissions due: March 7, 2013

WiAC '13: 2013 USENIX Women in Advanced Computing Summit
June 27, 2013

HotStorage '13: 5th USENIX Workshop on Hot Topics in Storage and File Systems
June 27–28, 2013
www.usenix.org/conference/hotstorage13
Submissions due: March 11, 2013

HotSWUp '13: 5th Workshop on Hot Topics in Software Upgrades
June 28, 2013
www.usenix.org/conference/hotswup13
Submissions due: March 7, 2013

USENIX Security '13: 22nd USENIX Security Symposium
August 14–16, 2013, Washington, DC, USA

Workshops Co-located with USENIX Security '13

EVT/WOTE '13: 2013 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections
August 12-13, 2013

CSET '13: 6th Workshop on Cyber Security Experimentation and Test
August 12, 2013
www.usenix.org/conference/cset13
Submissions due: April 25, 2013

HealthTech '13: 2013 USENIX Workshop on Health Information Technologies
Safety, Security, Privacy, and Interoperability of Health Information Technologies
August 12, 2013

LEET '13: 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats
August 12, 2013

FOCI '13: 3rd USENIX Workshop on Free and Open Communications on the Internet
August 13, 2013

HotSec '13: 8th USENIX Workshop on Hot Topics in Security
August 13, 2013

WOOT '13: 7th USENIX Workshop on Offensive Technologies
August 13, 2013

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Rikki Endsley
rikki@usenix.org

COPY EDITOR
Steve Gilmartin
proofshop@usenix.org

PRODUCTION
Arnold Gatilao
Casey Henderson
Michele Nelson

TYPESETTER
Star Type
startype@comcast.net

USENIX ASSOCIATION
2560 Ninth Street, Suite 215,
Berkeley, California 94710
Phone: (510) 528-8649
FAX: (510) 548-5738

www.usenix.org

;login: is the official magazine of the USENIX Association. *;login:* (ISSN 1044-6397) is published bi-monthly by the USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

$90 of each member's annual dues is for a subscription to *;login:*. Subscriptions for nonmembers are $90 per year. Periodicals postage paid at Berkeley, CA, and additional offices.

POSTMASTER: Send address changes to *;login:*, USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

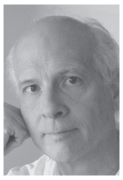
usenix ;login:

DECEMBER 2012, VOL. 37, NO. 6

OPINION

Musings

RIK FARROW



Rik is the editor of *;login:*.
rik@usenix.org

Almost all of the articles in this issue have to do with security—not surprising, as this is the annual security issue of *;login:*. Our first article, about protecting open source kernels from exploits hosted in user space, brought to mind something completely different, yet related: the design of the *Titanic*. The *Titanic* was billed as the fastest, safest, most modern ship of its era, but we all know what happened. Hubris, current construction techniques (brittle metals), and bad design all led to her sinking. What I want briefly to focus on are the watertight compartments that were supposed to prevent the *Titanic* from sinking. The watertight compartments were on the lowest level of the *Titanic*, but the barrier walls did *not* reach to the ceiling. As the bow sank, water overtopped the walls, speeding the *Titanic*'s, and 1502 people's, demise.

Compartments

Modern computers also have compartments, arranged via several related mechanisms: memory management, protection rings, and the system trap instruction. Memory management controls the access of regions of memory by user processes. To execute memory in the kernel, the system trap instruction transfers control to kernel code. And while in the kernel, code executes at ring 0, the most privileged ring.

Code executed with ring 0 has access to all memory. This access allows the kernel to read or write from anywhere in user space, so the kernel can copy data to devices or write the results of system calls into the memory of user processes. But executing code in user space while the CPU is in ring 0 is something that never needs to occur, and shouldn't happen. But it can, and does.

In Kemerlis et al., our lead article, the authors describe exploits that involve executing code in user space while still operating in ring 0. That gives the executed code the same access as the kernel itself, allowing an exploit to modify data structures in the kernel, and even to add new code in the form of back doors, keystroke loggers, and network monitors. Like the *Titanic*, the "watertight" compartments in most CPUs don't really work.

Kemerlis et al. have developed a compiler plugin, kGuard, that watches for attempts to jump into code in user space and blocks that from happening. Their method works for both Linux and BSD-based kernels today, and although the default remedy is a kernel panic, it does prevent the exploit from succeeding.

Intel, with its Supervisor Mode Execution Protection (SMEP) appearing in CPUs starting with Ivy Bridge, detects when the CPU attempts to execute code in user space while the CPU is in ring 0. The operating system has to enable SMEP during booting, and clearing a bit in register CR4 disables it. There are already articles about how to disable this feature in Windows 8 [1, 2], and some information about bypassing it in Linux [3]. Note that PaX [4] really prevents this type of attack, by arranging system memory using segment registers so kernel and user-space memory are truly separated.

Hardware

SMEP provides some hardware support for defending against return-to-user-space exploits. Perhaps if the ability to turn off SMEP could be disabled physically, the current exploit paths would be prevented. SMEP could work like the auditing feature in Linux and BSD kernels, where running `auditctl -e 2` prevents auditing from being changed or disabled without rebooting. I imagine that SMEP must be disabled during the initial boot process, and may remain disabled in some operating systems—ones that don't care about security.

Hardware is, well, hard. Implementing a proposed design change in silicon actually takes years. And once it is present, vendors, such as Intel and AMD, do not want the new feature to turn out to be a mistake. Still, I'd like to see even bigger changes than SMEP in future CPUs, with the ability to have much stronger compartmentalization high on my list.

We already have manycore CPUs, but these CPUs all share the same memory and use the same three hardware security mechanisms we have been using in computers since the 1960s. These hardware features—particularly how memory is managed—do a poor job of supporting microkernels, where only a small kernel runs in ring 0, as in seL4 [5]. Communication within current CPUs is via memory, and moving between regions of memory owned by different processes involves the intervention of the operating system, via both a system trap and a context switch. Of course, processes can decide to share memory regions, but then they need to manage the shared memory themselves. What has been missing is hardware support for message passing.

I'm aware that this is a difficult problem to solve, for many reasons: first, it is not how CPUs have been designed; second, because creating such a design would be revolutionary, it would require all new operating systems and applications. Or would it?

seL4 already can support a Linux API, and MINIX 3 [6] supports POSIX. As for running existing applications on top of very different hardware and operating systems, we have been doing this for years, via virtualization. There are also projects such as Microsoft's Drawbridge, designed to run untrusted applications within their own environment (a picoprocess) while providing as much of the required software stack (Windows libraries and the WinNT API) as needed [7]. Although Drawbridge is designed for today's CPUs, it could run much more securely in hardware-supported compartments. I suspect that open source operating systems could explore similar directions to support existing applications in hardware-supported sandboxes as well.

And even exascale computers (see Knauerhase et al. [8]) could benefit from these changes. In their design, most processors run execution engines, whereas only a few provide operating systems support via control engines. Although I don't know whether this design matches what I have long wanted, it does allow programmers to manipulate *data blocks* as first-class objects, and it might be a huge step toward an architecture that supports the message passing and strong compartmentalization I keep hoping for.

The Lineup

As mentioned, Kemerlis et al. explain why their solution, kGuard, has become important in protecting systems. Their solution is relatively lightweight (less than 1% performance hit for common server applications) while providing a higher level of security for today's hardware and operating systems.

Antonakakis et al. take us off in a different direction by using DNS error responses to locate bot-infected PCs. Current bots often use algorithmically determined domain names as a method to prevent loss of control of their bots through a takeover of the botnet C&C server. Pleiades watches for unsuccessful DNS resolution requests and filters them using machine-learning algorithms to match bots to known malware, and to identify new versions of malware based on the patterns of domain names generated.

Ceron et al., part of the Brazilian CERT team, have been watching attacks against VoIP servers for over a year. They set up honeypots listening for SIP connections, and recorded what happens after a connection to their fake SIP servers. In their article, they both analyze the traffic they've collected and make suggestions for securing SIP servers.

Winter and Crandall explain how parts of the Great Firewall of China (GFC) function. Through their work with the Tor Project, they have learned a lot about how a state agent goes about censoring the Internet and, in particular, blocking access to both Tor routers and to bridges. Theirs is a fascinating story, as nothing is quite as simple as it might seem.

Shekhar et al. provide a bridge between programming and security. They present AdSplit, a method for hosting advertisements with Android apps that separates them from the applications that would otherwise have included them. For advertisers, AdSplit means that the activity they see has not been spoofed, or the advertising hidden. For users, AdSplit moves advertising into a separate environment, so they don't share the same access privileges/permissions as the apps they are associated with.

Crossing partially over into sysadmin, Ur et al. tell us about their research with passwords and password meters. They have calculated the resistance to cracking various sets of passwords created using different rules. They also tested a variety of measures, using thousands of test subjects, to see what forms of feedback work best in helping users pick good passwords—that is, ones highly resistant to guessing or cracking.

Slipping over into the realm of programming, while not really abandoning security, Jang et al. explain how their Python script, ReDeBug, finds code clones. If you have ever programmed, you have likely borrowed code from other programs and repurposed it. In their research, Jang et al. have found thousands of examples of clones in open source code, but their real focus is on code that has been patched but remains unpatched in places it has been copied to—sometimes for as long as 10 years.

David Blank-Edelman takes us on another Perl adventure, this time into the realm of machine translation. Using a Google app as the example, David shows us how to translate languages and use a (moderately) RESTful Web service using Perl libraries.

Dave Beazley explores Pandas, a Python library for data analysis. Dave shows us how to slice and dice large data structures using both Pandas and numpy, an underlying Python library that provides an array object.

Dave Josephson has discovered a new hammer, and is looking for things to pound. The hammer is Nagios XI, a new interactive interface for Nagios that does so much more than just present an interface for system monitoring. Dave is very pleased.

Robert Ferrell was intrigued by both the Great Firewall of China, and the security design considerations of the typical programmer. He views the GFC from on high, then explains, through a simple analogy, what's wrong with most of our software.

Elizabeth Zwicky has reviewed several books, ranging from homebrew forensics (like CSI in your kitchen), two books supporting new versions of OS X, a book about problem-solving for programmers, and another Scrum book. Hint: she doesn't like all of them. I contribute a review of a short book about LEDs for lighting.

The EVT/WOTE '12 summary from the USENIX Security Symposium appears in this issue as well. Actually, all of the summaries from the Security Symposium appear online. With this issue, summaries will appear online as soon as they have been edited, copyedited, and formatted, and although this does take time, it won't be the several months that we have to wait for the print issue of *;login:*. It also means I can develop more articles for each issue without being so concerned about running out of space.

Unlike the *Titanic*, if your kernel is exploited, you will not even notice it—at first. It won't be a bone-jarring crunch as an iceberg the size of a small town rips through the hull, under the waterline. Instead, a successful exploit will silently provide remote control of your system, from the level with total access. Only failed exploits (that crash your kernel) will be "easy" to notice.

I do hope that some day we will have compartments that are complete, and still provide the same performance that we have today. Who knows? Maybe we can have both better security and more performance with future CPU designs.

To see videos from USENIX Security '12, visit: https://www.usenix.org/conferences/multimedia.

Resources

[1] Windows 8 exploit bypassing SMEP: http://packetstormsecurity.org/files/116618/SMEP_overview_and_partial_bypass_on_Windows_8.pdf.

[2] Bypassing Intel SMEP on Windows 8 x64: http://blog.ptsecurity.com/2012/09/bypassing-intel-smep-on-windows-8-x64.html.

[3] Dan Rosenberg, "SMEP: What Is It, and How to Beat It on Linux": http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/.

[4] PaX: http://grsecurity.net/.

[5] Secure Microkernel Project (seL4): http://www.ertos.nicta.com.au/research/sel4/.

[6] MINIX 3: http://www.minix3.org/.

[7] Drawbridge: http://research.microsoft.com/en-us/projects/drawbridge/.

[8] Knauerhase, Cledat, Teller, "For Extreme Parallelism, Your OS Is Sooooo Last-Millennium," *;login:*, vol. 37, no. 5 (October 2012), USENIX Association.

kGuard
Lightweight Kernel Protection

VASILEIOS P. KEMERLIS, GEORGIOS PORTOKALIDIS, ELIAS ATHANASOPOULOS, AND ANGELOS D. KEROMYTIS

Vasileios Kemerlis is a PhD student in the Department of Computer Science at Columbia University. His research interests are mainly in software and systems security, with a focus on automated software hardening. vpk@cs.columbia.edu

Georgios Portokalidis is a postdoctoral researcher in the Department of Computer Science at Columbia University. He obtained his doctorate from the Vrije Universiteit in Amsterdam. His research interests are mainly around the area of systems security, but extend to network monitoring, operating systems, and virtualization technologies. porto@cs.columbia.edu

Elias Athanasopoulos holds a BS in physics from the University of Athens, and an MS and PhD from the University of Crete. He is currently a Marie Curie postdoctoral fellow with Columbia University. elathan@cs.columbia.edu

Angelos D. Keromytis is an Associate Professor of Computer Science at Columbia University. His research interests revolve around systems and software security and reliability. He received his PhD in 2001 from the University of Pennsylvania. angelos@cs.columbia.edu

Kernel exploits have become increasingly popular over the past several years. We have developed kGuard, a cross-platform system that defends the operating system (OS) against a widespread class of kernel attacks. We describe how these attacks work and how kGuard protects the kernel with only a small decrease in performance.

The OS kernel is becoming an attractive target for attackers. The rising number of kernel vulnerabilities discovered and reported attest to this (see Figure 1). The reasons behind this trend are numerous. First, the number of applications running (continuously) with administrative privileges has significantly decreased, meaning that an attacker compromising such programs remotely gains only limited power over the underlying system. Additionally, programs have become harder to exploit due to the various defense mechanisms already adopted by modern OSes, such as address space layout randomization and stack smashing protection. The most interesting reason is probably that vulnerabilities such as NULL pointer dereference bugs, which were thought to be impractical, hard to exploit, and had not received significant attention by the security community, can be used with ease against the kernel to gain elevated privileges. In fact, some researchers proclaimed 2009 as "the year of the kernel NULL pointer dereference flaw" [2]. Last, exploiting kernel bugs has the added benefit of allowing attackers to mask their presence on the compromised systems (e.g., by hiding processes or files).

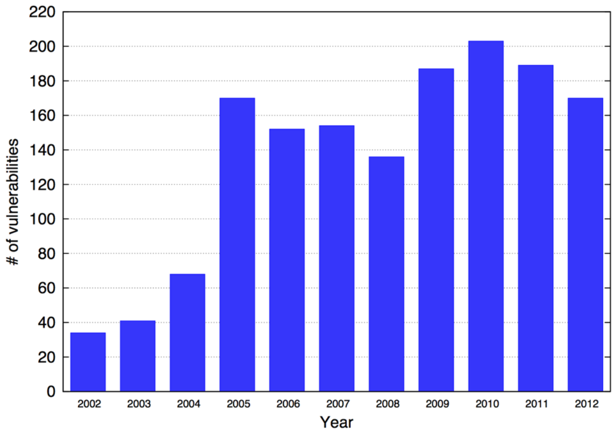


Figure 1: Kernel vulnerabilities (per year) reported to NIST. Over the past decade, the distinct number of CVE identifiers assigned to kernel vulnerabilities has increased by a factor of 5.

Kernel attacks are facilitated by the fact that user and kernel space (i.e., the memory area where user applications and the kernel reside, respectively), are weakly separated in modern OSes. As a result, direct transitions from more to less privileged protection domains (i.e., kernel to user space) are permissible, even though the reverse is not. This is what transforms NULL pointer dereference bugs from system instability vulnerabilities to privilege escalation threats. When exploited successfully, they enable local users to execute arbitrary code with kernel privileges, by redirecting the control flow of the kernel to user-controlled memory. Such return-to-user (ret2usr) attacks have affected all major OSes, including Windows, Linux, and the BSDs. These attacks are not limited to x86/x86-64 systems, but have also targeted the ARM, DEC, and PowerPC architectures.

Previous approaches to the problem are either impractical for deployment in certain environments or can be easily circumvented. For example, the most popular approach has been to disallow user processes to memory-map the lower part of their address space (i.e., the one including page zero). This scheme has been circumvented by various means and is not backwards compatible. The PaX [8] patch for x86 and x86-64 Linux kernels does not exhibit the same shortcomings, but greatly increases system call and I/O latency. Recent advances in virtualization have fostered a wave of research on extending virtual machine monitors (VMMs) to enforce the integrity of the virtualized guest kernels; however, virtualization is not always practical. Consider smartphone devices that use stripped-down versions of Linux and Windows, which are also vulnerable to such attacks. Running a complex VMM on current smartphones is not realistic due to their limited resources (i.e., CPU and battery life). On PCs, running the whole OS over a VM incurs performance penalties and management costs, while increasing the complexity and size of a VMM can introduce new bugs and vulnerabilities. Addressing the problem in hardware is the most efficient solution, but even though Intel has recently announced a new CPU feature, named SMEP [5], to thwart such attacks, hardware extensions are oftentimes adopted slowly by OSes. More importantly, other vendors have not publicly announced similar extensions.

kGuard is a lightweight solution to the problem. kGuard consists of a compiler plugin that augments kernel code with control-flow assertions, which ensure that privileged execution remains within its valid boundaries and does not cross to user space. This is achieved by identifying all exploitable control transfers during compilation, and injecting compact dynamic checks to attest that the kernel remains confined. kGuard is to some extent related to previous research on control-flow integrity (CFI) [1]; however, CFI is not effective against ret2usr attacks, because its integrity is only guaranteed if the attacker cannot overwrite the code of the protected binary or execute data. (During a ret2usr attack the control flow is redirected into memory pages whose contents and permissions are fully controlled by the attacker.)

Background

Virtual Memory Organization

Commodity OSes offer process isolation through private, hardware-enforced virtual address spaces; however, as they strive to squeeze more performance out of the hardware, they adopt a "shared" process/kernel memory model for minimizing the overhead of operations that cross protection domains, such as system calls, interrupts, and exceptions. Specifically, Windows and UNIX-like OSes divide virtual memory into user and kernel space. The former hosts user processes, while the latter holds kernel code and data, kernel extensions (modules), and device drivers. In most architectures, the separation between the two spaces is assisted and enforced by the following hardware features: CPU modes (or protection rings), a memory management unit (MMU), and special-purpose instructions. The x86/x86-64 instruction set architecture (ISA) supports four protection rings, with the kernel running in the most privileged one (ring 0) and user applications in the least privileged (ring 3). In fact, modern x86/x86-64 CPUs have more than four rings; hardware-assisted virtualization and System Management Mode are colloquially known as ring -1 and -2, respectively. Similarly, the PowerPC and MIPS platforms have two CPU modes, SPARC has three, and ARM seven. All these architectures feature an MMU, typically programmed using privileged special-purpose instructions, which implements virtual memory and ensures that memory assigned to a certain ring is not accessible by the less privileged ones.

Kernel Exploitation

Code running in user space cannot directly access or jump into the kernel, and hence, special-purpose instructions and hardware facilities (i.e., interrupts and exceptions) are provided for crossing the user/kernel boundary. Nevertheless, while executing privileged code, complete and unrestricted access to all memory and system objects is available. For example, when servicing a system call for a process (or during interrupt/exception handling) the kernel executes within the context of a preempted process and can directly access user memory to store the result of the call or read user data.

At the same time, OS kernels, which are mostly written in type-unsafe languages and assembly, suffer the same software flaws that plague applications. For instance, buffer and integer overflows, pointer arithmetic bugs, use-after-free vulnerabilities, and signedness errors can all be exploited to corrupt kernel memory and hijack control flow, thus executing arbitrary code with elevated privileges. The ability to trigger such a bug in the kernel, from a local process, provides a unique standpoint to attackers who totally control (i.e., both in terms of permissions and contents) part of the address space available to the kernel at any given time. In other words, "shellcode" can be executed with kernel rights by hijacking a privileged execution path and redirecting it to user space.

ret2usr Attacks

ret2usr attacks have become the most popular kernel exploitation method, for which a wealth of defensive mechanisms exists [7, 8, 5]. They are manifested by overwriting kernel data with user space addresses, after exploiting memory safety bugs in kernel code. As expected, attackers typically aim for control data [10], such as return addresses, jump tables, and function pointers, since these facilitate arbitrary code execution; however, pointers to critical data structures, frequently stored in kernel stack or heap, are also favored targets, since their contents can be tampered with by mapping fake copies in user space [9]. Most exploits of that kind target data structures that contain function pointers, or data that affect kernel execution, so as to diverge the control flow to arbitrary (typically user-controlled) places.

The end effect of these attacks is that the kernel is hijacked and control is redirected to user space code. Typically, ret2usr exploits use a multi-stage shellcode, where the first stage lies in user space and glues together kernel functions (i.e., the second stage shellcode) that perform privilege escalation or execute a rootshell. We refer to this type of exploitation as return-to-user [7] because it resembles the older return-to-libc [4] technique that redirected control to existing code in the C library. ret2usr attacks are yet another incarnation of the confused deputy problem [6], where a user fools the kernel (deputy) into misusing its authority and executing arbitrary, non-kernel code with elevated privileges.

kGuard

kGuard consists of a cross-platform GCC plugin that enforces address space segregation without relying on special hardware or architecture-specific features [8, 5]. It protects the kernel from ret2usr attacks with low-overhead by building on the following security primitives: *inline monitoring* and *code diversification*.

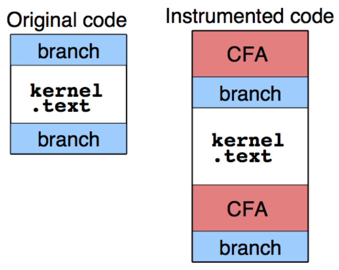


Figure 2a: CFA-based confinement. The injected guards perform a small runtime check before each computed branch to verify that the target address is in kernel space.

```
           cmp  $0xc0000000,%ebx
           jae  call_lbl
           mov  $0xc05af8f1,%ebx
call_lbl:  call *%ebx
```

Figure 2b: CFA$_R$ guard gets applied on an indirect `call` in x86 Linux (drivers/cpufreq/cpufreq.c).

Inline Monitoring

kGuard augments exploitable control transfers, at compile time, with dynamic control-flow assertions (CFAs) that, at runtime, prevent the unconstrained transition of privileged execution paths to user space. Figure 2a illustrates the concept. The injected CFAs perform a small runtime check before each indirect branch to verify that the target address is always in kernel space. If the assertion is true, execution continues normally, while if it fails because of a violation, execution is transferred to a handler that was inserted during compilation. The default handler appends a warning message to the kernel log and halts the system; however, custom handlers are also supported for facilitating forensic analysis (e.g., dumping the shellcode for studying new ret2usr exploitation vectors), selective confinement (i.e., avoiding instrumenting "legitimate" boundary violators such as VMware's I/O back door), and providing protection against persistent threats.

CFA guards come in two flavors, namely CFA$_R$ and CFA$_M$, depending on whether the protected control transfer uses a register or memory operand. Figure 2b shows an example of a CFA$_R$ guard. The code is from the `show()` routine of the cpufreq driver in x86 Linux. kGuard instruments the computed branch (`call *%ebx`) with three additional instructions. First, the `cmp` instruction compares the `ebx` register with the lower bound kernel address 0xC0000000. The same is also true for x86 FreeBSD/NetBSD (OpenBSD maps the kernel to the upper 512 MB of the virtual address space, and hence, its base address in x86 CPUs is located at 0xD0000000), whereas for x86-64 the check should be with address 0xFFFFFFFF80000000. In case the assertion is true, the control transfer is authorized by jumping to the `call` instruction. Otherwise, the `mov` instruction loads the address of the violation handler (0xC05AF8F1; `panic()`) into the branch register and proceeds to execute the `call`, which will invoke the violation handler.

Similarly, CFA_M guards confine indirect branches that use memory operands; however, these guards not only assert that the branch target is within the kernel address space, but also ensure that the memory address where the branch target is loaded from is also in kernel space. The latter is necessary for protecting against cases where attackers have managed to tamper with data structures that contain control data, by overwriting data pointers to such structures with user space addresses and mapping fake copies in user space. Interested readers are referred to our recent USENIX Security paper for more information regarding the CFA_M guards [7].

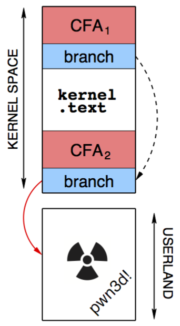


Figure 3: Subverting kGuard using bypass trampolines. CFA_1 succeeds since the address of the second branch (trampoline) is in kernel space. CFA_2 is completely bypassed by jumping directly to the branch instruction.

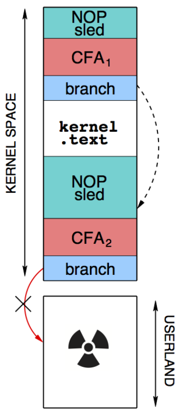


Figure 4: Code inflation reshapes the kernel's text area by inserting NOP sleds of random length at the beginning of each CFA.

Code Diversification

CFA_R and CFA_M guards provide reliable protection against ret2usr attacks only if the attacker exploits a kernel bug that allows him partially to control a computed branch target (e.g., by zeroing out certain bytes); however, vulnerabilities where the attacker can overwrite kernel memory with arbitrary values also exist [3]. When such flaws are present, exploits could attempt to bypass kGuard.

BYPASS TRAMPOLINES

To subvert kGuard, an attacker must be able to determine the address of a (indirect) control transfer instruction inside the text segment of the kernel. Moreover, she should also be able to control the value of its operand reliably (i.e., its branch target). We refer to that branch as a *bypass trampoline*. Note that in ISAs with overlapping variable-length instructions, finding an embedded opcode sequence that translates directly to a control branch in user space is possible. By overwriting the value of a protected branch target with the address of a bypass trampoline, the attacker can successfully execute a jump to user space, as depicted in Figure 3. The first CFA corresponding to the initially exploited branch will succeed, since the address of the trampoline remains inside the privileged memory segment, while the second CFA that guards the bypass trampoline is completely bypassed by jumping directly to the branch instruction.

CODE INFLATION

This technique reshapes the kernel's text area (see Figure 4). kGuard begins with randomizing the starting address of the text segment. This is achieved by inserting a random NOP sled at its beginning, which effectively shifts all executable instructions by an arbitrary offset. Next, it continues by inserting NOP sleds of random length at the beginning of each CFA. The end result is that the location of every computed control transfer instruction is randomized, making it harder for an attacker to guess the exact address of a confined branch to use as a bypass trampoline. The effects of the sleds are cumulative because each one pushes all instructions and NOP sleds following to higher memory addresses. The size of the initial sled is chosen by kGuard based on the target architecture.

The per-CFA NOP sled is randomly selected from a user-configured range. By specifying the range, users can trade higher overhead (both in terms of space and speed) for a smaller probability that an attacker can reliably obtain the address of a bypass trampoline. An important assumption of the aforementioned technique is the secrecy of the kernel's text and symbols. If the attacker has access to the binary image of the confined kernel or is armed with a kernel-level memory leak, the probability of successfully guessing the address of a bypass trampoline increases; however, assigning safe file permissions to the kernel's text, modules, and debugging symbols is not a limiting factor. This can be trivially achieved by changing the permissions in the file system to disallow reads, from non-administrative users, in `/boot` and `/lib/modules` in Linux/FreeBSD, `/bsd` in OpenBSD, etc. In fact, this is considered standard practice in OS hardening, and is automatically enabled in PaX and similar patches, as well as in the latest Ubuntu Linux releases. Also note that the kernel should harden access to the system message ring buffer (`dmesg`) and certain files in the `proc` pseudo-file system, in order to prevent the leakage of kernel addresses.

CFA MOTION

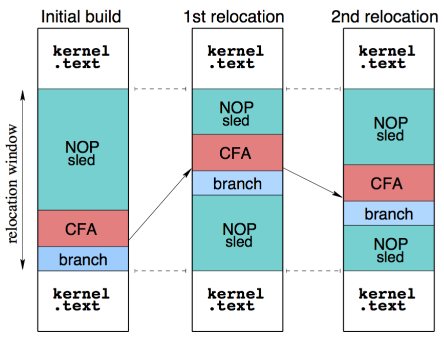


Figure 5: CFA motion synopsis. kGuard relocates each inline guard and protected branch, within a certain window, by routinely rewriting the text segment of the kernel.

The basic idea behind this technique is the "continuous" relocation of the protected branches and injected guards, by rewriting the text segment of the kernel, for more hardening against bypasses. Figure 5 illustrates the concept. During compilation, kGuard emits information regarding each injected CFA, which can be used later to relocate the respective code snippets. Specifically, kGuard logs the exact location of the CFA inside the kernel's text, the type and size of the guard, the length of the prepended NOP sled, as well as the size of the protected branch. Armed with that information, kGuard can then migrate every CFA and indirect branch instruction separately, by moving it inside the following window: `sizeof (nop_sled) + sizeof (cfa) + sizeof (branch)`. Currently, kGuard only supports CFA motion during kernel bootstrap. That said, keep in mind that ret2usr violations are detected at runtime, and hence one false guess is enough to identify the attacker and restrict his capabilities (e.g., by revoking his access to prevent brute-force attempts).

Results and Next Steps

The effectiveness of kGuard has been experimentally assessed by instrumenting different vanilla Linux kernels, both in x86 and x86-64 architectures, and testing them against real exploits that cover a broad spectrum of different flaws, including direct NULL pointer dereferences, control hijacking via tampered data structures (data pointer corruption), function and data pointer overwrite, arbitrary kernel-memory nullification, and ret2usr via kernel stack-smashing. As expected, kGuard was able to detect and prevent exploitation successfully in all cases. For more information regarding the evaluation suite, please refer to our paper in USENIX Security '12 [7].

kGuard exhibits lower overhead than previous work. On average, it imposes a 11.4% overhead on system call and I/O latency on x86 Linux, and 10.3% on x86-64, as reported by the LMbench micro-benchmark suite. In the case of IPC bandwidth, it exhibits an average slowdown of 6% on x86, and 6.6% on x86-64. Additionally, the size of a kGuard-compiled kernel grows between 3.5% and 5.6%, due to the inserted checks, while the impact on real-life applications, such as the MySQL RDBMS and Apache Web server, is minimal ($\leq$ 1%).

Future steps include investigating how to apply the CFA motion technique while a kernel is running and the OS is live. Currently, we have developed a Linux prototype that utilizes a dedicated kernel thread, which upon a certain condition, freezes the kernel and performs rewriting. Thus far, we have achieved CFA relocation in a coarse-grained manner by exploiting the suspend subsystem of the Linux kernel. Specifically, we bring the system to pre-suspend state to prevent any kernel code from being invoked during relocation (note that the BSD OSes have similar capabilities); however, our end goal is to perform CFA motion in a more fine-grained, non-interruptible and efficient manner, without "locking" the whole OS. Further in the future, we also plan to explore custom fault handlers that perform error virtualization for automatically recovering from attacks.

Conclusion

kGuard is a fast and flexible cross-platform solution that protects the kernel from ret2usr attacks. It works by injecting fine-grained inline guards during the translation phase that are resistant to bypass, and does not require any modification to the kernel or additional software such as a VMM. kGuard can safeguard both 32- and 64-bit OSes that map a mixture of code segments with different privileges inside the same scope and are vulnerable to ret2usr exploits. We believe that kGuard strikes a balance between safety and functionality, and provides comprehensive protection from a widespread class of attacks.

Availability

kGuard is freely available at: http://www.cs.columbia.edu/~vpk/research/kguard/.

Acknowledgments

We thank Georgios Kontaxis for his valuable feedback on earlier drafts of this manuscript. This work was supported by DARPA, the US Air Force, and ONR through Contracts DARPA-FA8750-10-2-0253, AFRL-FA8650-10-C-7024, and N00014-12-1-0166, respectively. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, the Air Force, or ONR.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," *Proceedings of the 12th ACM Conference on Computer and Communications Security* (CCS), 2005, pp. 340–353.

[2] M.J. Cox, "Red Hat's Top 11 Most Serious Flaw Types for 2009," February 2010: http://www.awe.com/mark/blog/20100216.html.

[3] CVE-2010-3904, October 2010: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3904.

[4] S. Designer, "Getting Around Non-Executable Stack (and Fix)," August 1997: http://seclists.org/bugtraq/1997/Aug/63.

[5] V. George, T. Piazza, and H. Jiang, "Technology Insight: Intel Next Generation Microarchitecture Codename Ivy Bridge," September 2011: www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf.

[6] N. Hardy, “The Confused Deputy (or Why Capabilities Might Have Been Invented),” . SIGOPS Operating Systems Review, vol. 22, no. 4, October 1988, pp. 36–38.

[7] V.P. Kemerlis, G. Portokalidis, and A.D. Keromytis, “kGuard: Lightweight Kernel Protection Against Return-to-User Attacks,” *Proceedings of the 21st USENIX Security Symposium,* USENIX Association, 2012, pp. 459–474.

[8] PaX Team home page: http://pax.grsecurity.net, accessed September 2012.

[9] SecurityFocus, “Linux Kernel ‘pipe.c’ Local Privilege Escalation Vulnerability,” November 2009: http://www.securityfocus.com/bid/36901/info.

[10] Virtual Security Research, “Linux RDS Protocol Local Privilege Escalation,” October 2010: http://www.vsecurity.com/resources/advisory/20101019-1/.

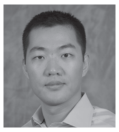
Detecting and Tracking the Rise of DGA-Based Malware

MANOS ANTONAKAKIS, ROBERTO PERDISCI, NIKOLAOS VASILOGLOU, AND WENKE LEE

Manos Antonakakis received his PhD in computer science from Georgia Institute of Technology under Professor Wenke Lee's supervision. Currently, he works at Damballa as the Director of Academic Sciences where he is responsible for academic research, university collaborations, and technology transfer efforts. His main research interests are in network security and machine learning/data mining. manos@damballa.com

Roberto Perdisci is an Assistant Professor in the Computer Science Department at the University of Georgia, and an Adjunct Assistant Professor in the School of Computer Science at the Georgia Institute of Technology. He is the recipient of a 2012 NSF CAREER Award. His main research interests are in network security and machine learning/data mining. perdisci@cs.uga.edu

Nikolaos Vasiloglou received his PhD in electrical engineering from the Georgia Institute of Technology. He has extensive experience in developing machine-learning applications and algorithms. In the past he has developed machine-learning engines and models for several companies. nvasil@ieee.org



Wenke Lee is a Professor in the School of Computer Science at Georgia Institute of Technology and the Director of the Georgia Tech Information Security Center (GTISC). He earned his PhD in computer science from Columbia University in 1999. He has published more than 100 scholarly articles. His current research projects are in the areas of botnet detection, malware analysis, virtual machine monitoring, and Web 2.0 security and privacy, with funding from NSF, DHS, DoD, and industry. wenke@cc.gatech.edu

When bots go in search of their command and control (C&C) servers, they often use algorithmically generated domain names (DGAs). We have created a system (Pleiades) that watches unsuccessful DNS resolution requests (NXDomain) from recursive DNS servers in large networks. Pleiades can reliably identify new clusters of NXDomains generated by DGAs, the newly infected hosts, and often, the actual C&C servers the DGA malware employs. In this article, we explain how our system works, as well as the most interesting information about current bot infections and C&C structures.

Introduction

Botnets are groups of malware-compromised machines, or bots, that can be remotely controlled by an attacker (the botmaster) through a command and control (C&C) communication channel. Botnets have become the main platform for cyber-criminals to send spam, steal private information, host phishing Web pages, etc. Over time, attackers have developed C&C channels with different network structures. Most botnets today rely on a centralized C&C server, whereby bots query a predefined C&C domain name that resolves to the IP address of the C&C server from which commands will be received. Such centralized C&C structures suffer from the "single point of failure" problem because if the C&C domain is identified and taken down, the botmaster loses control over the entire botnet.

To overcome this limitation, attackers have used P2P-based C&C structures in botnets such as Nugache, Storm, and more recently, Waledac, Zeus, and Alureon (aka TDL4). Whereas P2P botnets provide a more robust C&C structure that is difficult to detect and take down, they are typically harder to implement and maintain. In an effort to combine the simplicity of centralized C&Cs with the robustness of P2P-based structures, attackers have recently developed a number of botnets that locate their C&C server through automatically generated pseudo-random domains names. In order to contact the botmaster, each bot periodically executes a domain generation algorithm (DGA) that, given a random seed (e.g., the current date), produces a list of candidate C&C domains. The bot then attempts to resolve these domain names by sending DNS queries until one of the domains resolves to the IP address of a C&C server. This strategy provides a remarkable level of agility because even if one or more C&C domain names or IP addresses are identified and taken down, the bots will eventually get the IP address of the relocated C&C server via DNS queries to the next set of automatically generated domains. Notable examples of DGA-based botnets (or DGA-bots, for short) are Bobax, Kraken, Sinowal (aka Torpig), Srizbi, Conficker-A/B/C, and Murofet.

A defender can attempt to reverse engineer the bot malware, particularly its DGA algorithm, to pre-compute current and future candidate C&C domains in order to detect, block, and even take down the botnet; however, reverse engineering is not always feasible because the bot malware can be updated very quickly (e.g., hourly) and obfuscated (e.g., encrypted, and only decrypted and executed by external triggers such as time).

We recently proposed a novel detection system, called Pleiades [1], capable of identifying DGA-bots within a monitored network without reverse engineering the bot malware. Pleiades is placed between the network machines and the local recursive DNS (RDNS) server (aka "below" the recursive DNS infrastructure of the network) or simply at the edge of a network to monitor DNS query/response messages from/to the machines within the network. Specifically, Pleiades analyzes DNS queries for domain names that result in Name Error responses, also called "NXDOMAIN" responses, i.e., domain names for which no IP addresses (or other resource records) exist.

The focus on NXDomains is motivated by the fact that modern DGA-bots tend to query large sets of domain names among which relatively few successfully resolve to the IP address of the C&C server. Therefore, to identify DGA domain names automatically, Pleiades searches for relatively large clusters of NXDomains that (1) have similar syntactic features and (2) are queried by multiple potentially compromised machines during a given epoch.

The intuition is that in a large network, such as the ISP network where we ran our experiments, multiple hosts may be compromised with the same DGA-bots. Therefore, each of these compromised assets will generate several DNS queries resulting in NXDomains, and a subset of these NXDomains will likely be queried by more than one compromised machine. Pleiades automatically is able to identify and filter out "accidental" user-generated NXDomains due to typos or misconfigurations. When Pleiades finds a cluster of NXDomains, it applies statistical learning techniques to build a model of the DGA. This is used later to detect future compromised machines running the same DGA and to detect active domain names that "look similar" to NXDomains resulting from the DGA and therefore probably point to the botnet C&C server's address.

Overview of Pleiades

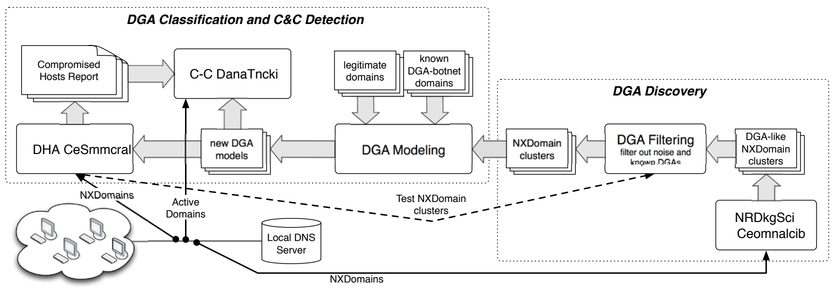


Figure 1: A high-level overview of Pleiades

Next, we provide a high-level overview of our DGA-bot detection system, Pleiades. As shown in Figure 1, Pleiades consists of two main modules: a DGA Discovery module and a DGA Classification and C&C Detection module. We discuss the roles of these two main modules and their components, and how they are used in coordination to learn actively and update DGA-bot detection models.

DGA Discovery

The DGA Discovery module analyzes streams of unsuccessful DNS resolutions, as seen from "below" a local DNS server (see Figure 1). All NXDomains generated by network users are collected during a given epoch (e.g., one day). Then, the collected NXDomains are clustered according to the following two similarity criteria: (1) the domain name strings have similar statistical characteristics (e.g., similar length, level of "randomness," character frequency distribution, etc.), and (2) the domains have been queried by overlapping sets of hosts. The main objective of this NXDomain clustering process is to group together domain names that likely are automatically generated by the same algorithm running on multiple machines within the monitored network.

Naturally, because this clustering step is unsupervised, some of the output NXDomain clusters may contain groups of domains that happen to be similar by chance (e.g., NXDomains due to common typos or to misconfigured applications). Therefore, we apply a subsequent filtering step. We use a supervised DGA Classifier to prune NXDomain clusters that appear to be generated by DGAs that we have previously discovered and modeled, or that contain domain names that are similar to popular legitimate domains. The final output of the DGA Discovery module is a set of NXDomain clusters, each of which likely represents the NXDomains generated by previously unknown or not yet modeled DGA-bots.

DGA Classification and C&C Detection

Every time a new DGA is discovered, we use a supervised learning approach to build models of what the domains generated by this new DGA "look like." In particular, we build two different statistical models: (1) a statistical multi-class classifier that focuses on assigning a specific DGA label (e.g., DGA-Conficker.C) to the set of NXDomains generated by a host h_i and (2) a hidden Markov model (HMM) that focuses on finding single active domain names queried by h_i that are likely generated by a DGA (e.g., DGA-Conficker.C) running on the host, and are therefore good candidate C&C domains.

The DGA Modeling component receives different sets of domains labeled as Legitimate (i.e., "non-DGA"), DGA-Bobax, DGA-Torpig/Sinowal, DGA-Conficker.C, New-DGA-v1, New-DGA-v2, etc., and performs the training of the multi-class DGA Classifier and the HMM-based C&C Detection module.

The DGA Classification module works as follows. Similar to the DGA Discovery module, we monitor the stream of NXDomains generated by each client machine "below" the local recursive DNS server. Given a subset of NXDomains generated by a machine, we extract a number of statistical features related to the NXDomain strings. Then we ask the DGA Classifier to identify whether this subset of NXDomains resembles the NXDomains generated by previously discovered DGAs. That is, the classifier will either label the subset of NXDomains as generated by a known DGA, or tell us that it does not fit any model. If the subset of NXDomains is assigned a specific DGA label (e.g., DGA-Conficker.C), the host that generated the NXDomains is deemed to be compromised by the related DGA-bot.

Once we obtain the list of machines that appear to be compromised with DGA-based bots, we take the detection one step further. While all previous steps focused on NXDomains, we now turn our attention to domain names for which we observe valid resolutions. Our goal is to identify which domain names, among the ones generated by the discovered DGA-based bots, actually resolve into a valid IP address. In other words, we aim to identify the botnet's active C&C server.

To achieve this goal, we consider all domain names that are successfully resolved by hosts that have been classified as running a given DGA, say New-DGA-vX, by the DGA Classifier. Then we test these successfully resolved domains against an HMM specifically trained to recognize domains generated by New-DGA-vX. The HMM analyzes the sequence of characters that compose a domain name d, and computes the likelihood that d is generated by New-DGA-vX.

DGA Discoveries and Case Studies

In this section, we present the most important experimental results of our system. We will elaborate on the DGAs we discovered throughout the two years of NXDomain monitoring period at a large US ISP. Then we will summarize the most interesting findings from the 13 DGAs we detected. Seven of them use a DGA algorithm from a known malware family. The other six, at the time of discovery and to the best of our knowledge, have no known malware association. We will conclude with three cases studies of currently active threats that employ DGAs for their C&C call-back communications.

New DGAs

Pleiades began clustering NXDomain traffic on November 1, 2010. We bootstrapped the DGA modeler with domain names from already known DGAs and also a set of Alexa domain names as the benign class. In Table 1, we present all unique clusters we discovered throughout the evaluation period. The "Malware Family" column simply maps the variant to a known malware family if possible. We discover the malware family by checking the NXDomains that overlap with NXDomains we extracted from traffic obtained from a malware repository. Also, we manually inspected the clusters with the help of a security company's threat

New-DGA-v1	New-DGA-v2	New-DGA-v3
71f9d3d1.net	clfnoooqfpdc.com	uwhornfrqsdbrbnbuhjt.com
a8459681.com	slsleujrrzwx.com	epmsgxuotsciklvywmck.com
a8459681.info	qzycprhfiwfb.com	nxmglieidfsdolcakggk.com
a8459681.net	uvphgewngjiq.com	ieheckbkkkoibskrqana.com
1738a9aa.com	gxnbtlvvwmyg.com	qabgwxmkqdeixsqavxhr.com
1738a9aa.info	wdlmurglkuxb.com	gmjvfbhfcfkfyotdvbtv.com
1738a9aa.net	zzopaahxctfh.com	sajltlsbigtfexpxvsri.com
84c7e2a3.com	bzqbcftfcrqf.com	uxyjfflvoqoephfywjcq.com
84c7e2a3.info	rjvmrkkycfuh.com	kantifyosseefhdgilha.com
84c7e2a3.net	itzbkyunmzfv.com	lmklwkkrficnnqugqlpj.com

New-DGA-v4	New-DGA-v5	New-DGA-v6
semk1cquvjufayg02orednzdfg.com	zpdyaislnu.net	lymylorozig.eu
invfgg4szr22sbjbmdqm51pdtf.com	vvbmjfxpyi.net	lyvejujolec.eu
0vqbqcuqdv0i1fadodtm5iumye.com	oisbyccilt.net	xuxusujenes.eu
np1r0vnqjr3vbs3c3iqyuwe3vf.com	vgkblzdsde.net	gacezobeqon.eu
s3fhkbdu4dmc00ltmxskleeqrf.com	bxrvftzvoc.net	tufecagemyl.eu
gup1iapsm2xiedyefet21sxete.com	dlftozdnxn.net	lyvitexemod.eu
y5rk0hgujfgo0t4sfers2xolte.com	gybszkmpse.net	mavulymupiv.eu
me5oclqrfano4z0mx4qsbpdufc.com	dycsmcfwwa.net	jenokirifux.eu
jwhnr2uu3zp0ep40cttq3oyeed.com	dpwxwmkbxl.net	fotyriwavix.eu
ja4baqnv02qoxlsjxqrszdziwb.com	ttbkuogzum.net	vojugycavov.eu

Figure 2: A sample of 10 NXDomains for each DGA cluster that we could not associate with a known malware family

Malware Family	First Seen	Population on Discovery
Shiz/Simda-C [8]	03/20/11	37
Bamital [4]	04/01/11	175
BankPatch [2]	04/01/11	28
Expiro.Z [3]	04/30/11	7
Boonana [9]	08/03/11	24
Zeus.v3 [7]	09/15/11	39
TDSS/TDL DGA Variant	07/08/12	201
New-DGA-v1	01/11/10	12
New-DGA-v2	01/18/11	10
New-DGA-v3	02/01/11	18
New-DGA-v4	03/05/11	22
New-DGA-v5	04/21/11	5
New-DGA-v6	11/20/11	10

Table 1: DGAs detected by Pleiades

team. The "First Seen" column denotes the first time we saw traffic from each DGA variant. Finally, the "Population on Discovery" column shows the variant population on the discovery day. We can see that we can detect each DGA variant with an average number of 32 "infected hosts" across the entire statewide ISP network coverage.

As we see in Table 1, Pleiades reported seven variants that belong to known DGA-enabled malware families [2–4, 7–9]. Six more variants of NXDomains were reported and modeled by Pleiades, but for these, to the best of our knowledge, no known malware can be associated with them. A sample set of 10 NXDomains for each one of these variants can be seen in Figure 2.

Within a two-year period of our experiments, we observed an average population of 742 Conficker-infected hosts in the ISP network. Murofet had the second largest population of infected hosts at 92 per day, while the Boonana DGA came in third with an average population of 84 infected hosts per day. The fastest growing DGA was Zeus.v3 with an average population of 50 hosts per day, but during the last four days of the experiments, the Zeus.v3 DGA had an average of 134 infected hosts. It is worth noting the New-DGA-v1 had an average of 19 hosts per day, the most populous of the newly identified DGAs.

FALSE REPORTS ON NEW DGAS

During our evaluation period we came across five categories of clusters falsely reported as new DGAs. In all of the cases, we modeled these classes in the DGA modeler as variants of the benign class. We now discuss each case in detail.

The first cluster of NXDomains falsely reported by Pleiades were random domain names generated by Chrome [10, 5]. Each time the Google Chrome browser starts, it will query three "random looking" domain names. These domain names are issued as a DNS check, so the browser can determine whether NXDomain rewriting is enabled. The "Chrome DGA" was reported as a variant of Bobax from Pleiades. We trained a class for this DGA and flagged it as benign. One more case of testing for NXDomain rewriting was identified in a brand of wireless access points: Connectify offers wireless hot-spot functionality, and one of their configuration options enables the user to hijack the ISP's default NXDomain rewriting service. The device generates a fixed number of NXDomains to test for rewriting.

Two additional cases of false reports were triggered by domain names from the `.it` and `.edu` TLDs. These domain names contained minor variations on common words (i.e., repubblica, gazzetta, computer, etc.). Domain names that matched these clusters appeared only for two days in our traces and never again. The very short-lived presence of these two clusters could be explained if the domain names were part of a spam campaign that was remediated by authorities before it became live.

The fifth case of false report originated from domain names under a US government zone and contained the string `wpdhsmp`. Our best guess is that these are internal domain names that were accidentally leaked to the recursive DNS server of our ISP. Domain names from this cluster appeared only for one day. This class of NXDomains was also modeled as a benign variant. It is worth noting that all falsely reported DGA clusters, excluding the Chrome cluster, were short-lived. If operators are willing to wait a few days until a new DGA cluster is reported by Pleiades, these false alarms would not have been raised.

Case Studies

Next we discuss the three most interesting active threats that employ DGA techniques as part of their C&C life cycle.

ZEUS.V3

In September 2011, Pleiades detected a new DGA that we linked to the Zeus.v3 variant a few weeks later. The domain names collected from the machines compromised by this DGA-malware are hosted in six different TLDs: `.biz`, `.com`, `.info` , `.net` , `.org`, and `.ru`. Excluding the top-level domains, the length of the domain names generated by this DGA are between 33 and 45 alphanumeric characters. By analyzing one sample of the malware, we observed that its primary C&C infrastructure is P2P-based. If the malware fails to reach its P2P C&C network, it follows a contingency plan, where a DGA-based component is used to try to recover from the loss of C&C communication. The malware will then resolve pseudo-random domain names, until an active C&C domain name is found.

To date, we have discovered 12 such C&C domains. Over time, these 12 domains resolved to five different C&C IPs hosted in four different networks: three in the US (AS6245, AS16626, and AS3595) and one in the United Kingdom (AS24931). Interestingly, we observed that the UK-based C&C IP address remained active for only a few minutes, from Jan 25, 2012 12:14:04 EST to Jan 25, 2012 12:22:37 EST. The C&C moved from a US IP (AS16626) to the UK (AS24931), and then almost immediately back to the US (AS3595).

BANKPATCH

We picked the BankPatch DGA cluster as a sample case for analysis because this botnet had been active for several months during our experiments and the infected population continues to be significant. The C&C infrastructure that supports this botnet is impressive. Twenty-six different clusters of servers acted as the C&Cs for this botnet. The botnet operators not only made use of a DGA but also moved the active C&Cs to different networks every few weeks (on average). During our C&C discovery process, we observed IP addresses controlled by a European CERT. This CERT has been taking over domain names from this botnet for several months. We managed to cross-validate with them the completeness and correctness of the C&C infrastructure. Complete information about the C&C infrastructure can be found in Table 2.

IP Addresses	CC	Owner
146.185.250.{89-92}	RU	Petersburg Int.
31.11.43.{25-26}	RO	SC EQUILIBRIUM
31.11.43.{191-194}	RO	SC EQUILIBRIUM
46.16.240.{11-15}	UA	iNet Colocation
62.122.73.{11-14,18}	UA	"Leksim" Ltd.
87.229.126.{11-16}	HU	Webenlet Kft.
94.63.240.{11-14}	RO	Com Frecatei
94.199.51.{25-18}	HU	NET23-AS 23VNET
94.61.247.{188-193}	RO	Vatra Luminoasa
88.80.13.{111-116}	SE	PRQ-AS PeRiQuito
109.163.226.{3-5}	RO	VOXILITY-AS
94.63.149.{105-106}	RO	SC CORAL IT
94.63.149.{171-175}	RO	SC CORAL IT
176.53.17.{211-212}	TR	Radore Hosting
176.53.17.{51-56}	TR	Radore Hosting
31.210.125.{5-8}	TR	Radore Hosting
31.131.4.{117-123}	UA	LEVEL7-AS IM
91.228.111.{26-29}	UA	LEVEL7-AS IM
94.177.51.{24-25}	UA	LEVEL7-AS IM
95.64.55.{15-16}	RO	NETSERV-AS
95.64.61.{51-54}	RO	NETSERV-AS
194.11.16.133	RU	PIN-AS Petersburg
46.161.10.{34-37}	RU	PIN-AS Petersburg
46.161.29.102	RU	PIN-AS Petersburg
95.215.{0-1}.29	RU	PIN-AS Petersburg
95.215.0.{91-94}	RU	PIN-AS Petersburg
124.109.3.{3-6}	TH	SERVENET-AS-TH-AP
213.163.91.{43-46}	NL	INTERACTIVE3D-AS
200.63.41.{25-28}	PA	Panamaserver.com

Table 2: C&C infrastructure for BankPatch

CIDR	CC	Owner
146.185.250.0/24	RU	PIN-AS
83.133.0.0/16	EU	LAMBDANET-AS
195.3.144.0/22	LV	RN-DATA-LV
94.63.149.0/24	RO	CORAL-IT
194.11.16.0/24	RU	PIN-AS
94.63.240.0/24	RO	POSTOLACHE
188.95.48.0/21	NL	GLOBALLAYER
46.251.224.0/20	DE	WEBTRAFFIC
95.215.0.0/22	RU	PIN-AS
94.60.122.0/23	RO	COVER-SUN-DESIGN
109.236.80.0/20	NL	WORLDSTREAM
63.223.96.0/19	US	JOVITA
91.212.226.0/24	RU	ZHIRK
46.161.28.0/23	RU	PIN-AS
141.136.16.0/24	RO	SC-MORE-SECURE-SRL
46.249.32.0/19	NL	SERVERIUS-AS
217.23.0.0/20	NL	WORLDSTREAM
62.122.74.0/23	EU	ROOT SA
50.7.192.0/19	US	FDCSERVERS
38.0.0.0/8	US	COGENT Cogent/PSI
194.247.182.0/23	UA	UDNET
195.234.124.0/22	UA	KOSMOTEL
195.28.10.0/23	RU	Neryungrinskoye
89.208.144.0/20	RU	DINET-AS
94.228.208.0/20	NL	NETROUTING-AS
27.255.64.0/19	KR	LGDACOM
91.199.75.0/24	DE	INTEROUTE
120.197.80.0/20	CN	CMNET

Table 3: Extended criminal network infrastructure behind New TDSS/TDL4 DGA variant

The actual structure of the domain name used by this DGA can be separated into a four-byte prefix and a suffix string argument. The suffix string arguments we observed were: `seapollo.com`, `tomvader.com`, `aulmala.com`, `apontis.com`, `fnomosk.com`, `erhogeld.com`, `erobots.com`, `ndsontex.com`, `rtehedel.com`, `nconnect.com`, `edsafe.com`, `berhogeld.com`, `musallied.com`, `newnacion.com`, `susaname.com`, `tvolveras.com`, `dminmont.com`, `esroater.com`, `jierihon.com` and `mobama.com`.

The four bytes of entropy for the DGA were provided by the prefix. We observed collisions between NXDomains from different days, especially when only one suffix argument was active. Therefore, we registered a small sample of 10 domain names at the beginning of 2012 in an effort to obtain a glimpse of the overall distribution of this botnet. Over a period of one month of monitoring the sinkholed data from the domain name of this DGA, this botnet has infected hosts in 270 different networks distributed across 25 different countries. By observing the recursive DNS servers from the domain names we sinkholed, we determined 4,295 were located in the US. The recursives we monitored were part of this list, and we were able to measure 86 infected hosts (on average) in the network we were monitoring. The five countries that had the most DNS resolution requests for the sinkholed domain names (besides the US) were Japan, Canada, the United Kingdom, and Singapore. The average number of recursive DNS servers from th ese countries that contacted our authorities was 22, significantly smaller than the volume of recursive DNS servers within the US.

TDSS/TDL4 DGA VARIANT

This TDSS/TDL4 DGA variant is the latest DGA discovery made possible by Pleiades. At the time of this writing, no malware sample has been discovered for this DGA variant. We believe that the DGA is primarily used to serve traditional C&C and enables click-fraud activities for the main TDSS/TDL4 [6] infection. This new DGA variant for TDSS/TDL4 appeared as a new DGA cluster in the beginning of July 2012. The C&C network-hosting infrastructure spans multiple different networks in Europe, US, and Asia. While most of the C&C IP addresses have been associated in the past with illicit operations (i.e., RBN, BitCoin mining) and have affected hundreds of thousands of victims, we are not aware of a sample available to the security community that resembles the DGA's behavior.

In an effort to describe the extended criminal C&C network for this TDSS/TDL4 variant, we first obtained the C&C domain names and remote IPs from the successful DNS resolutions observed by the TDSS/TDL4 DGA victims. Then we projected them in our passive DNS data collection in order to discover their immediate related historic IPs and domain names. We then selected all the domain names that matched the HMM model for this DGA variant. The resulting set of resource records constitutes the extended TDSS/TDL4 C&C network. Using the RDATA extracted from our passive DNS, we can provide a complete picture of the extended C&C network components. In Table 3, we show the extended criminal network behind this threat. We were able to identify 85 hosts that appear to be related to the actors behind TDSS/TDL4 DGA and that were used over the past 18 months.

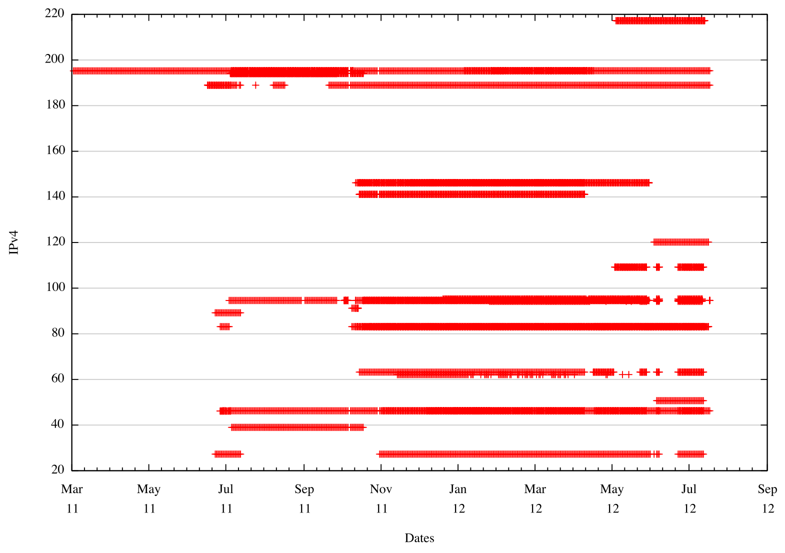


Figure 3: TDSS/TDL4 network agility

In Figure 3, we observe the network agility of the extended TDSS/TDL4 C&C network infrastructure and note how the botmasters behind TDSS/TDL4 moved and updated their impressive C&C network infrastructure from 03/03/2011 through 07/18/2012. Multiple C&Cs hosts were clearly active at the same time, especially towards the last few months of our analysis period.

0sso151a47nztrxld6.com	**ntibwlwhgl6bjp.com**
0ubpccgkvzrnng.com	**nxadrmwfgplgpr72jcv.com**
1jndmf93bnobmbm-v.com	**qiqxqoedngojyw4d.com**
3tvcqyg4msj9byffzm.com	**sbvv2b59psvpghaojz9.com**
4rtgobtumvrzoqwq.com	**smaug.gtisc.gatech.edu**
ad9btvkonim6wsgg8lv.com	**t407bqgh56jbkv4ua.com**
anz7sjg6awufloz.com	**udf-szhubujmuhp1jj.com**
cudkkm05bzvn0dth.com	**v-qk5nvogztncpmg2cp.com**
d8kkkblaj6c4olp.com	**vlxbbxrhq1llnft.com**
dklfebjexiabttkwvgos.com	**vpmybkeogu4vfuiu5s.com**
fjg56xwoupqpdxr.com	**wxbppgbdwiedzbnzhh.com**
fwjudokrkhlwd3sm.com	**xrqc-swswrykw30p.com**
hrai41zpyw73sxhja5k3.com	**yhftaw6wxlrhbl90osg.com**
ikh9w-3vdmlndafja.com	**ymgn1thqfe4q6rs.com**
nihawelnopjmpn67yrn.com	**zv7dfcgtusnttpl.com**

Figure 4: TDSS/TDL4 DGA NXDomain samples

In Figure 4, we present a small sample from the NXDomains the TDSS/TDL4 DGA generated over time. A few new NXDomains appear to be generated by the infected hosts every 48 hours. Using this observation, and in collaboration with Georgia Tech Information Security Center (GTISC), we managed to get a glimpse of the botnet worldwide infection levels. As of September 15, 2012, we have observed more than 250,000 unique Internet hosts tying to contact the GTISC sinkhole. Unfortunately, this number is growing, which implies that either the infection campaign is still active or the threat is largely undetectable by traditional network and host level defenses.

Conclusion

With this short article, we summarize the key aspects of a novel detection system called Pleiades. This system is able to detect machines accurately within a monitored network that are compromised with DGA-based bots. Utilizing the streams of unsuccessful DNS resolution from a large ISP, Pleiades can identify and model previously unknown DGAs, instead of relying on manual reverse engineering of bot malware and their DGA algorithms. In our multi-month evaluation phase, Pleiades was able to identify seven DGAs that belong to known malware families and six new DGAs never reported before.

References

[1] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon, "From Throw-Away Traffic to Bots: Detecting the Rise of DGA-Based Malware," 21st USENIX Security Symposium, USENIX Association, 2012.

[2] BankPatch, Trojan.Bankpatch.C: http://www.symantec.com/security_response/writeup.jsp?docid=2008-081817-1808-99, 2009.

[3] Microsoft Malware Protection Center, Virus:Win32/Expiro.Z: http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?Name=Virus%3AWin32%2FExpiro.Z, 2011.

[4] M. Geide, "Another Trojan Bamital Pattern": http://research.zscaler.com/2011/05/another-trojan-bamital-pattern.html, 2011.

[5] S. Krishnan and F. Monrose, "DNS Prefetching and Its Privacy Implications: When Good Things Go Bad," *Proceedings of the 3rd USENIX Conference on Large-Scale Exploits and Emergent Threats* (LEET '10), USENIX Association, 2010, pp. 10-10.USENIX Association.

[6] A. Matrosov, E. Rodionov, and D. Harley, TDSS parts 1 through 3: http://resources.infosecinstitute.com/tdss4-part-1/, http://resources.infosecinstitute.com/tdss4-part-2/, http://resources.infosecinstitute.com/tdss4-part-3/, 2011.

[7] CERT Polska, "ZeuS P2P+DGA Variant Mapping Out and Understanding the Threat": http://www.cert.pl/news/4711/langswitch_lang/en, 2012.

[8]Sophos, Mal/Simda-C: http://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Mal~Simda-C/detailed-analysis.aspx, 2012.

[9] Microsoft Malware Protection Center, Trojan:Java/Boonana: http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?Name=Trojan%3AJava%2FBoonana, 2011.

[10] B. Zdrnja, "Google Chrome and (Weird) DNS Requests": http://isc.sans.edu/diary/Google+Chrome+and+weird+DNS+requests/10312, 2011.

Anatomy of SIP Attacks

JOÃO M. CERON, KLAUS STEDING-JESSEN, AND CRISTINE HOEPERS

João Marcelo Ceron is a Security Analyst at CERT.br/NIC.br. He holds a master's degree from Federal University of Rio Grande do Sul, Brazil, where he worked with honeypots for botnet detection. He currently works with incident handling, and his research interests include honeypot data analysis.
ceron@cert.br

Klaus Steding-Jessen is CERT.br/NIC.br Technical Manager. He holds a PhD in applied computing from the Brazilian National Institute for Space Research, where he worked with honeypots for spam detection. His research area is the use of honeypots for detecting Internet infrastructure abuse by attackers and spammers. He is also one of the authors of the chkrootkit tool.
jessen@cert.br

Cristine Hoepers is CERT.br/NIC.br General Manager. She holds a PhD in applied computing from the Brazilian National Institute for Space Research in the area of honeypot data analysis. Her research interests include the use of honeypots for incident detection and trend analysis. She has spoken at several security conferences, including FIRST, APWG, MAAWG, LACNIC, and AusCERT. cristine@cert.br

In the past few years we have seen a steady increase in the popularity of VoIP (Voice over IP) services. Scans for SIP (Session Initiation Protocol [4]) servers have been reported for many years, and to gather more details about these activities we emulated SIP servers in a network of 50 low-interaction honeypots, and collected data about these attacks for 358 days. What will follow is a description of our observations and advice on how to prevent these attacks from being successful.

Tracking SIP Servers Abuse

For quite some time, the security community has been reporting an increase in scans for the SIP default port 5060/UDP, as well as some anecdotal evidence of other types of abuse. Similarly, at the CERT.br honeyTARG Honeynet Project [3] (a chapter of The Honeynet Project), port 5060/UDP was consistently among the top-10 targeted ports. Bearing that in mind, we have been tracking the abuse of SIP servers more closely since last year.

This project consists of 50 low-interaction honeypots, based on Honeyd [7], deployed in the Brazilian Internet space. In order to enable Honeyd to collect SIP attack information, we implemented a listener that emulates Asterisk Server [2] and allows the definition of which extensions are available, as well as their default responses and passwords. This software allows us to collect the initial stages of a SIP session, logging information such as attack origins and the phone numbers the attackers attempted to call. For privacy reasons, we chose not to record audio sessions, limiting the implementation only to the SIP signaling.

Figure 1 presents a SIP conversation fragment logged by our listener. There are two SIP methods: `REGISTER` and `INVITE`. The first part is a `REGISTER` request. This is used by a user agent (UA) for registering contact information, such as its current IP address. The second part illustrates the `INVITE` method, which is used to establish a media session between UAs. In this log, a UA places a call from the extension 100 to the external phone number "201*****274" (sanitized number). Additionally, the "user-agent" field shows that this UA has provided the identification string "X-Lite release 1006e stamp 34025", a common softphone.

```
2011-12-27 19:48:38 +0000: sip-honeyd.pl[4429]: IP: 41.X.X.19,
method: REGISTER, from: "100", to: "100", resp: 200,
user-agent: "X-Lite release 1006e stamp 34025"

2011-12-27 19:48:39 +0000: sip-honeyd.pl[4429]: IP: 41.X.X.19,
method: INVITE, from: "100", to: "201*****274", resp: 486,
user-agent: "X-Lite release 1006e stamp 34025"
```

Figure 1: Honeypot log showing the attacker's IP, the phone number being requested, and the user agent identification string

Making Sense of the Data

The traffic targeted to port 5060/UDP in our honeypots was related to the following attack steps:

1. **Scanning**: searching for SIP servers.
2. **Enumeration**: once a SIP server is identified, the attackers try to enumerate the server configuration, available extensions, and so on.
3. **Brute force:** attackers try to access extensions that are protected with weak passwords.
4. **Abuse:** after gaining access to a PABX extension, the attackers will try to call external PSTN (Public Switched Telephone Network) numbers, usually to place international calls.

In a preliminary analysis of the collected data, we were able to identify that the attackers would try to call a given number by using several prefixes to increase the attack success (see Figure 2). This occurs because a SIP server can be configured in different ways—for example "0" or "9" to access PSTN lines. In some countries, such as Brazil, one must also specify the telecommunication operator to be used for long distance calls.

```
2011-09-26 19:21:55 -0300: sip2db.pl[9814]:  00149725*****586
2011-09-26 19:21:55 -0300: sip2db.pl[9814]: 000149725*****586
2011-09-26 19:21:55 -0300: sip2db.pl[9814]:  00159725*****586
2011-09-26 19:21:55 -0300: sip2db.pl[9814]:  00219725*****586
2011-09-26 19:21:55 -0300: sip2db.pl[9814]:  00219725*****586
2011-09-26 19:21:55 -0300: sip2db.pl[9814]:  00319725*****586
2011-09-26 19:21:55 -0300: sip2db.pl[9814]:      9725*****586
2011-09-26 19:21:55 -0300: sip2db.pl[9814]:  00219725*****586
                                                 <----------|
                                                 9725*****586
```

Figure 2: A phone number requested in different ways in order to identify the correct prefix for dialing long distance or international calls

Figure 2 illustrates the many variations one attacker was using for the phone number "9725*****586". To deal with this situation, which we called redundancy, we implemented a heuristic to identify it and to store only a unified SIP session related to this number in the database. Besides reducing the size of the database, this heuristic also helped us to identify the phone number's country code and to correlate calls placed at different times and coming from different sources, to a unique phone number.

Table 1 summarizes the data that reached our honeypot infrastructure from September 2011 to September 2012.

Data	Count
REGISTER messages	64,249,923
INVITE messages	1,007,697
Unified INVITE messages	153,773
Unique IPs	7,752
Unique Autonomous System Numbers - ASNs	858
Total number of days	358
Unique source country codes - CCs	83

Table 1: Summary of the data collected from September 2011 to September 2012

The majority of the `REGISTER` messages are from automated scans. Most of them have the signature of the SipVicious toolkit [5], a collection of tools for auditing SIP-based VoIP systems. The `INVITE` messages are actual abuse attempts directed to the listeners, i.e., phone call attempts. The unified `INVITE` messages are the `INVITE` messages after redundancies were identified. Note that the number of unique ASNs and CCs demonstrate a high dispersion of the origin of the attacks.

In the following sections, we will focus on the analysis of the unified `INVITE` messages, including the phone numbers that were called the most and the abuse sources.

User Agents and IDS Evasion

An important piece of information logged is the user agent identification string provided by the SIP clients that connected to the honeypots and tried to place a call. The most frequent user agents provided are presented in Figure 3.

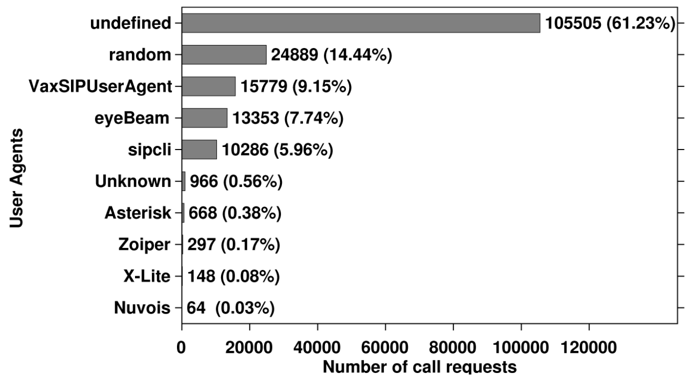


Figure 3: Top user agent identification strings provided by the SIP clients that tried to place calls

Note that 61.23% of the connections came from SIP clients that didn't provide any user agent string. We grouped all these clients under an identification string we called "undefined." This behavior is not expected from SIP clients and may suggest that many attackers are using customized tools to abuse SIP servers. Also note that the third most frequent user agent is the "VaxSIPUserAgent," which is used by a software development kit, also suggesting customized tools.

Additionally, there was a group of attempts where the user agent almost never repeated. In every new session, the client provided a random 20-character user agent, as shown in Table 2. This behavior was the second most frequent and was observed even in sequential requests coming from the same IP address. Our best guess is that this is being used to hide attack fingerprints or to evade IDS detection.

Timestamp	IP	User Agent String
2012-01-23T04:02:15Z	194.X.X.131	DmQCAsNRKZYayfosaXES
2012-01-23T04:02:17Z	194.X.X.131	yy3BHtWnCBPco3knmRqG
2012-01-23T04:02:19Z	194.X.X.131	KdUhQNVVxaZYfHg0rXFD
2012-01-23T04:02:21Z	194.X.X.131	otYvAff8mpZviS2CfF6M
2012-01-23T04:02:23Z	194.X.X.131	5y5ttWMXPbFIeyHb4l4D
2012-01-23T04:02:25Z	194.X.X.131	YDjb3Q8Wiw6442YCXMnE

Table 2: Examples of random user agent identification strings captured by the honeypots

We have also observed user agents commonly used by SIP servers, such as Asterisk. These user agents could be fake (set by the attacker) or could represent compromised SIP servers used to abuse other servers. The remaining user agents presented in Figure 3 refer to popular softphones.

As we can see, almost 85% of all connections came from customized or potentially malicious software.

Where Is It Coming From?

When looking into the source of the abuse attempts, we can try to identify specific patterns in the geographical origin and try to identify other characteristics that could give some insights about possible motivations.

Based on the source IP addresses of the attempted calls, we were able to estimate the source country code (CC) for the attacks. The country code allocation is based on information provided by the Regional Internet Registries (RIRs). Figure 4 shows these top CCs.

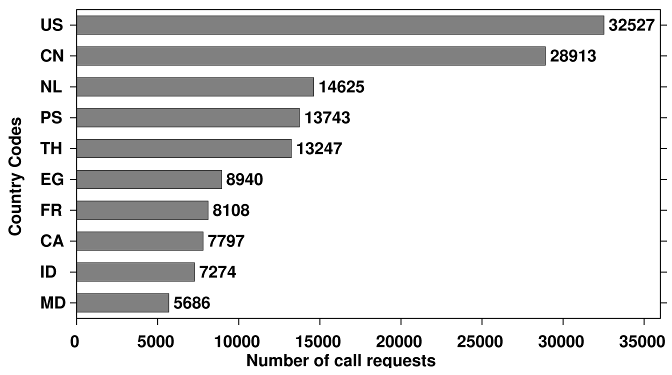


Figure 4: Top country codes of call requests (based on source IPs), aggregated by requested phone numbers

In Table 3 we list the top 10 IP addresses. For each IP, there is also information about how many different IDD (International Direct Dialing) codes it tried to call and the user agent string provided.

#	Count	IP	CC	IDDs	User Agent String
01	19,562	113.X.X.205	CN	142	undefined
02	11,027	83.X.X.16	NL	62	undefined
03	7,553	71.X.X.9	US	67	VaxSIPUserAgent/3.0
04	7,486	50.X.X.99	US	27	undefined
05	6,412	49.X.X.93	TH	38	undefined
06	5,647	85.X.X.212	FR	2	undefined
07	5,343	122.X.X.83	TH	37	undefined
08	4,672	24.X.X.37	CA	6	undefined
09	4,640	194.X.X.36	MD	48	random
10	4,234	202.X.X.204	ID	27	undefined

Table 3: Top source IPs, country codes, number of countries called, and the user agent provided

The most frequent CCs observed are US and CN, which are also the ones for three of the unique IPs that tried to place most of the calls. Note that the top 10 IPs were responsible for 44% of all call attempts. Additionally, the first IP address is responsible for 67% of all attempts coming from Chinese IPs. Likewise, the third and fourth IPs were responsible for 47% of all call attempts coming from IPs allocated to the US.

Another interesting fact is that the user agents provided by the top source IPs are not those of popular softphones but, instead, are possibly from customized attack tools. And, most interestingly, all the user agents provided by the ninth IP were 20-character random strings, as discussed in the previous section.

This combination of few IPs with distinctive user agents points to the possibility of these being rogue VoIP servers or proxies used as hubs to place phone calls.

Considering that one of the expected behaviors of a rogue VoIP server is high geographic dispersion of the destination phone calls, we tried to corroborate this hypothesis with additional analysis. We used AfterGlow [1] to explore the relationship among the top IP (113.X.X.205) and the destination of all calls. Figure 5 presents this IP address and the country codes it attempted to call. The CC was determined using the Perl library Number::Phone::Country, that associates an IDD to a country code. We can see that this IP, a possible VoIP server, places calls to 142 different countries.

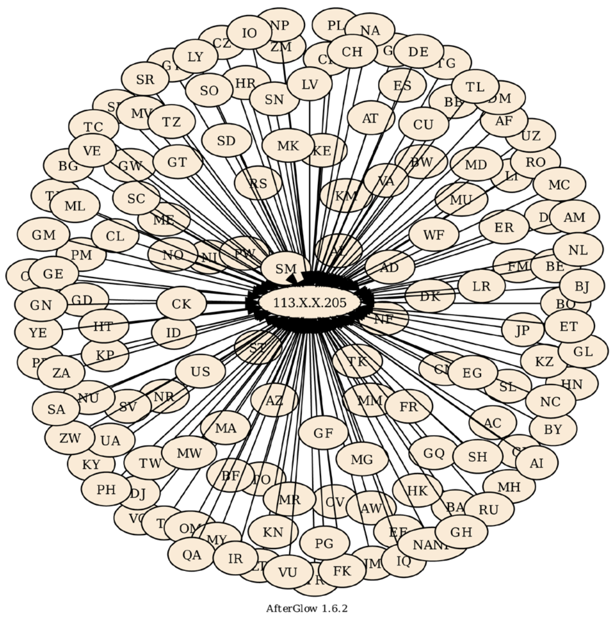


Figure 5: Destination country code for all calls placed by the IP 113.X.X.205

Who They Are Calling, and Why...

To gain more understanding of the abuse, we have also studied the nature of the phone numbers the attackers attempted to call. The most requested phone numbers fall into the following categories:

- **Cell phones:** identified by the number prefix
- **Financial services:** customer services from financial institutions (mainly Bank of America and Citibank)
- **Pre-paid phones:** pre-paid card services for international phone calls (Net2Phone)
- **Customer relations:** e-commerce customer relation services

The most called phone number was Bank of America's Credit Card Customer Service, totaling 5,090 attempts. Only seven IPs requested this phone number, and four of them have the rogue VoIP server behavior that was discussed earlier. Actually, 64% of all calls to Bank of America's Credit Card Customer Service came from the eighth IP listed in Table 3.

Source IP	Destination IDD	Count	(%)
PS	IL	7305	4.23%
EG	EG	6138	3.56%
MD	CZ	5559	3.22%
US	CZ	4535	2.63%
FR	RU	4264	2.47%
CA	800 (Free)	3296	1.91%
US	IL	2088	1.21%
US	ZW	1904	1.10%
CA	CZ	1903	1.10%
DE	CZ	1749	1.01%

Table 4: Most frequent combinations of source IPs and destination IDD country codes

The IP addresses that originated the calls were, for the most part, not the same as the IDD destination country. Table 4 shows the most frequent pairs, consisting of IPs allocated to a country code that are calling numbers in a given destination IDD country code.

Based on the data analyzed so far, we can present some hypotheses about the attackers' motivations:

1. Abusing SIP servers in order to place free phone calls or to gain anonymity;
2. Abusing the premium-rate telephone numbers business model;
3. Reselling VoIP services by abusing poorly configured SIP servers; and
4. Validating personal identifiable information, such as credit cards and bank account details.

Securing Your SIP Server

The types of activities observed reinforce the importance of implementing the current SIP security best practices [6]. Most attacks would have been prevented or mitigated by following one or more of these recommendations:

- **Protect the SIP server from the Internet:** be more restrictive in terms of which extensions can be reached from external IP addresses.
- **Use strong passwords:** use long, hard-to-guess passwords. Most SIP clients require the password to be entered only once, so there is no need to create easy-to-remember passwords. The current recommendation is to use at least 12-character passwords, including numbers, symbols, and lower and uppercase letters.
- **Create usernames different from extensions:** most brute force attempts try usernames that match the extension numbers.
- **Monitor the SIP use in your organization:** monitor your SIP server logs for abuse attempts, but also keep an eye on your PSTN billing information, looking for unusual long distance and international calls.

Conclusion

As the adoption of SIP services grows, being aware of the characteristics of the abuse against them is increasingly important. As our analysis showed, almost 85% of all call requests came from customized or potentially malicious software, and some of the calls may be related to unlawful activities. Also, because there are attackers currently taking advantage of poorly configured servers, the need to increase monitoring is clear.

The good news is that the implementation of basic VoIP security best practices will prevent most of the attacks seen in the wild.

References

[1] AfterGlow: Link Graph Visualization: http://afterglow.sourceforge.net/.

[2] Asterisk: The Open Source Telephony Projects: http://www.asterisk.org/.

[3] CERT.br: honeyTARG Honeynet Project: http://honeytarg.cert.br/.

[4] RFC 3261—SIP: Session Initiation Protocol: http://www.ietf.org/rfc/rfc3261.txt.

[5] SIPVicious—Tools for auditing SIP-based VoIP systems: http://blog.sipvicious.org/.

[6] John Todd, "Seven Steps to Better SIP Security with Asterisk," 2009: http://blogs.digium.com/2009/03/28/sip-security/.

[7] Niels Provos and Thorsten Holz, *Virtual Honeypots: From Botnet Tracking to Intrusion Detection,* Addison-Wesley, 2008.

AdSplit
Separating Smartphone Advertising from Applications

SHASHI SHEKHAR, MICHAEL DIETZ, AND DAN S. WALLACH

Shashi Shekhar graduated with an MS in computer science from Rice University and is a software engineer at Google.
shashi.iitg@gmail.com

Michael Dietz is a PhD student at Rice University.
mdietz@gmail.com

Dan S. Wallach is a Professor of Computer Science at Rice University.
dwallach@cs.rice.edu

A wide variety of smartphone applications today rely on third-party advertising services, which provide libraries that are linked into the hosting application. Advertising libraries often need additional permissions, requiring applications to issue requests for additional permissions to their users at install time. This article describes our AdSplit model, where we extended Android to allow an application and its advertising to run as separate processes, under separate user IDs, eliminating the need for applications to request permissions on behalf of their advertising libraries.

Introduction

The smartphone and tablet markets are growing in leaps and bounds, helped in no small part by the availability of specialized third-party applications ("apps"). Whether on the iPhone or Android platforms, apps often come in two flavors: a free version, with embedded advertising, and a pay version without. Both models have been successful in the marketplace. To pick one example, the popular Angry Birds game at one point brought in roughly equal revenue from paid downloads on Apple iOS devices and from advertising-supported free downloads on Android devices [1]. They now offer advertising-supported free downloads on both platforms.

We cannot predict whether free or paid apps will dominate in the years to come, but advertising-supported apps will certainly remain prominent. Already, a cottage industry of companies offer advertising services for smartphone app developers.

Today, these services are simply pre-compiled code libraries, linked and shipped together with an app. This means that a remote advertising server has no way to validate a request it receives from a user legitimately clicking on an advertisement. A malicious app could easily forge these messages, generating revenue for its developer while hiding the advertisements in their entirety. To create a clear trust boundary, advertisers would benefit from running ads separately from their host apps.

In Android, apps must request permission at install time for any sensitive privileges they want to exercise. Such privileges include access to the Internet, access to coarse or fine location information, or even access to see what other apps are installed on the phone. Advertisers want this information in order to better profile users and thus target ads at them; in return, advertisers may pay more money to their hosting apps' developers. Consequently, many apps that require no particular permissions, by themselves, suffer permission bloat, being forced to request the privileges required by their advertising libraries in addition to any of their own needed privileges. Because users might be scared away by detailed permission requests, app developers would also benefit if ads could be hosted in separate apps, which might then make their own privilege requests or be given a suitable one-size-fits-all policy.

Finally, separating apps from their advertisements creates better fault isolation. If the ad system fails or runs slowly, the host apps should be able to carry on without inconveniencing the user. Addressing these needs requires developing a suitable software architecture, with OS assistance to make it robust.

This article primarily focuses on the current state of practice in the Android marketplace, giving a flavor of how we engineered AdSplit as a proof of concept for a better system design.

App Analysis

The need to monetize freely distributed smartphone applications has given rise to many different ad provider networks and libraries. The companies competing for business in the mobile ad world range from established Web ad providers, such as Google's AdMob, to a variety of dedicated smartphone advertising firms.

With so many options for serving mobile ads, many app developers choose to include multiple ad libraries. Additionally, there is a new trend of advertisement aggregators that have the aggregator choose which ad library to use in order to maximize profits for the developer.

Although we're not particularly interested in advertising market share, we want to understand how these ad libraries behave. What permissions do they require? And how many apps would be operating with fewer permissions, if only their advertisement systems didn't require them? To address these questions, we downloaded approximately 10,000 free apps from the Android Market and the Amazon App Store and analyzed them.

Permissions Required

Every ad library requires Internet access, presumably to download the ad content to be displayed. Many libraries want additional privileges to assist in customizing ads. This ranges from location information to the ability to see what else is running on your phone. Presumably, better targeted ads will bring greater revenue to the application developer.

Permission Bloat

In Android, an application requests a set of permissions at the time it's installed. Those permissions must suffice for all of the app's needs and for the needs of its advertising library. We decided to measure how many of the permissions requested are used exclusively by the advertising library (i.e., if the advertising library were removed, the permission would be unnecessary).

Our results, shown in Figure 1, are quite striking: 15% of apps requesting Internet permissions are doing so for the sole benefit of their advertising libraries; 26% of apps requesting coarse location permissions are doing it for the sole benefit of their advertising libraries; and 47% of apps requesting permission to get a list of the tasks running on the phone (the ad libraries use this to check whether the applica-

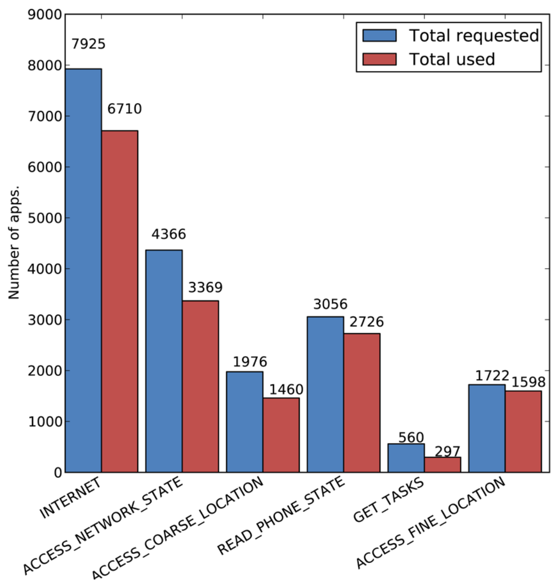


Figure 1: Distribution of types of permissions reduced when advertisements are separated from applications

tion hosting the advertisement is in foreground) are doing so for the sole benefit of their advertising libraries. These results suggest that any architecture that separates advertisements from applications will be able to reduce permission bloat significantly. (In concurrent work to our own, Grace et al. [5] performed a static analysis of 100,000 Android apps and found advertisement libraries uploading sensitive information to remote ad servers. They also found that some advertisement libraries were fetching and dynamically executing code from remote ad servers.)

Design Objectives

The first and most prominent design decision of AdSplit is to separate a host application from its advertisements. This separation has a number of ramifications:

- **Specification for advertisements.** Currently, the ad libraries are compiled and linked with their corresponding host application. If advertisements are separate, then the host activities must contain the description of which advertisements to use. We introduced a method by which the host activity can specify the particular ad libraries to be used.
- **Permission separation.** AdSplit allows advertisements and host applications to have distinct and independent permission sets.
- **Process separation.** AdSplit advertisements run in separate processes, isolated from the host application.
- **Life-cycle management.** Advertisements only need to run when the host application is running, otherwise they can be safely killed; similarly, once the host application starts running, the associated advertisement process must also start running. Our system manages the life cycle of advertisements.
- **Screen sharing.** Advertisements are displayed inside a host app, so if advertisements are separated, there should be a way to share screen real estate. AdSplit includes a mechanism for sharing screen real estate.
- **Authenticated user input.** Advertisements generate revenue for their host applications; this revenue is typically dependent on the amount of user interaction

with the advertisement. The host application can try to forge user input and generate fraudulent revenue, hence the advertisements should have a way to determine whether input events received from the host application are genuine. AdSplit includes a method by which advertising applications can validate user input, validate that they are being displayed on-screen, and pass that verification, in an unforgeable fashion, to their remote server.

The AdSplit Design

Because we want to factor out the advertising code into a separate process/activity, this will require a variety of changes to ensure that the user experience is unchanged.

An app using AdSplit will require the collaboration of three major components: the host activity, the advertisement activity, and the advertisement service. The host activity is the app that the user wants to run, whether a game, a utility, or whatever else. It then "hosts" the advertisement activity, which displays the advertisement. There is a one-to-one mapping between host activity and advertisement activity instances. The UNIX processes behind these activities have distinct user IDs and distinct permissions granted to them. To coordinate these two activities, we have a central advertisement service. The ad service is responsible for delivering UI events to the ad activity. It also verifies that the ad activity is being properly displayed and that the UI clicks aren't forged.

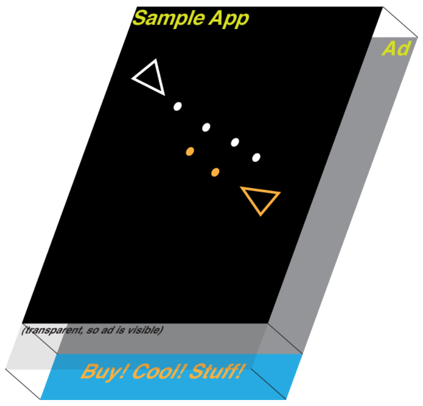


Figure 2: Screen sharing between host and advertisement apps

AdSplit builds on Quire [2], which prototyped a feature shown in Figure 2, allowing the host and advertisement activities to share the screen together. This Quire feature, when combined with a standard Android feature that allows the advertisement activity to detect when its UI is occluded, provides the underpinnings of AdSplit 's UI compositing system.

Permission Separation

With Android's install-time permission system, an application requests every permission it needs at the time of its installation. As we described above, advertising libraries cause significant bloat in the permission requests made by their hosting applications. Our AdSplit architecture allows the advertisements to run as separate Android users with their own isolated permissions. Host applications no longer need to request permissions on behalf of their advertisement libraries.

We note that AdSplit makes no attempt to block a host application from explicitly delegating permissions to its advertisements. For example, the host application might obtain fine-grained location permissions (i.e., GPS coordinates with meter-level accuracy) and pass these coordinates to an advertising library that lacks any location permissions. Plenty of other Android extensions, including TaintDroid [3] and Paranoid Android [8], offer information-flow mechanisms that might be able to forbid this sort of thing if it was considered undesirable. We believe these techniques are complementary to our own, but we note that if we cannot create a hospitable environment for advertisers, they will have no incentive to run in an environment like AdSplit.

Separation for Legacy Apps

A significant number of current apps with embedded advertising libraries would immediately benefit from AdSplit, reducing the permission bloat necessary to host embedded ads. This section describes a proof-of-concept implementation that can automatically rewrite an Android app to use AdSplit. Something like this could be deployed in an app store or even directly on the smartphone itself.

We first built a rewriting system that decompiled an Android app, replacing the internal advertising library with a stub that called out to our AdSplit advertising service. Although we got this working for one specific library, there are a number of problems that would stand in the way of this as a general-purpose solution for AdSplit:

Ad Installation

When advertisements exist as distinct apps in the Android ecosystem, they will need to be installed somehow. We're hesitant to give the host app the necessary privileges to install third-party advertising code. Perhaps an app could declare that it had a dependency on a third-party app, and the main installer could hide this complexity from the user, in much the same way that common Linux package installers will follow dependencies as part of the installation process for any given target.

Ad Permissions

Even if we can get the ad libraries installed, we have the challenge of understanding what permissions to grant them. Particularly when many advertising libraries know how to make optional use of a permission, such as measuring the smartphone's location if it's allowed, how should we decide if the advertisement app has those permissions? Unfortunately, there is no good solution here, particularly not without generating complex user interfaces to manage these security policies.

Ad Unloading

Like any Android app, an advertisement app must be prepared to be killed at any time—a consequence of Android's resource management system. This could have some destabilizing consequences if the hosting app is trying to communicate with its advertisement and the ad is killed. Also, what happens if a user wants to uninstall an advertising app? Should that be forbidden unless every host app which uses it is also uninstalled?

For further details about the implementation of AdSplit 's legacy app support and automatic rewriting, please see our full paper [9].

Alternative Design: HTML Ads

While struggling with the shortcomings outlined above, we hit upon an alternative approach that uses the same AdSplit architecture. The solution is to expand on something that advertising libraries are already doing: embedded Web views.

Ad creators purchasing advertising on smartphones will want to specify their advertisements the same way they do for the Web: as plain text, images, or perhaps as a "rich" ad using JavaScript. Needless to say, a wide variety of tools are available to produce such ads, and mobile advertising providers want to make it easy for ads to appear on any platform (iPhone, Android, etc.) without requiring heroic effort from the ad creators.

Consequently, all of the advertising libraries we examined simply include a WebView within themselves. Most of the native Android advertising code is really nothing more than a wrapper around a WebView. Based on this insight, we suggest that it will be easiest to deploy AdSplit by providing a single advertising app, built into the Android core distribution, that satisfies the typical needs of Android advertising vendors.

Installation becomes a non-issue, since the only advertiser-provided content in the system is HTML, JavaScript, and/or images. We still use the rest of the AdSplit architecture, running the WebView with a separate user ID, in a separate process and activity, ensuring that a malicious app cannot tamper with the advertisements it hosts.

Security permissions are more straightforward. The same-origin policy, standard across the entire Web, applies perfectly to HTML AdSplit. Since the Android WebView is built on the same Webkit browser as the real Web browser app, it has the same security machinery to enforce the same-origin policy.

Keeping all this in mind, we built a new form of WebView specifically targeted for HTML ads: the AdWebView. The AdWebView is a way to host HTML ads in a constrained manner. We introduced two advertisement-specific permissions that can be controlled by the user. These permissions control whether ads can make Internet connections or use geolocation features of HTML5.

When an ad inside an AdWebView requests to load a URL or performs a call to the HTML5 geolocation API, the AdWebView performs a permission check to verify whether the associated advertisement origin has the needed advertisement permission. These advertisement permissions can be managed by the user in exactly the same way they are for any other Web pages.

About the only open policy question is whether we should allow AdSplit HTML advertisements to maintain long-term tracking cookies or whether we should disable any persistent state. Certainly, persistent cookies are a standard practice for Web advertising, so they seem like a reasonable feature to support here as well. AdWebView, by default, doesn't support persistent cookies, but it would be trivial to add.

Policy

Although AdSplit allows for and incentivizes applications to run separately from their advertisements, there are a variety of policy and user experience issues that we must still address.

Advertisement Blocking

Once advertisements run as distinct processes, some fraction of Android users will see this as an opportunity to block advertisements for good. Certainly, with Web browsers, AdBlock and AdBlock Plus are incredibly popular. The Chrome Web store lists these two extensions in its top six with "over a million" installs each. (Google doesn't disclose exact numbers.)

The Firefox add-ons page offers more details, claiming that AdBlock Plus is far and away the most popular Firefox extension, having been installed just over 14 million times, versus 7 million for the next most popular extension. The Mozilla Foundation estimates that 85% of their users have installed an extension (http://blog.mozilla.com/addons/2011/06/21/firefox-4-add-on-users/). Many will install an ad blocker.

To pick one example, Ars Technica, a Web site popular with tech-savvy users, estimated that about 40% of its users ran ad blockers [7]. At one point, it added code to display blank pages to these users in an attempt to cajole them into either paying for ad-free "premium" service, or at least configuring their ad blocker to "white list" the Ars Technica Web site.

Strategies such as this are perilous. Some users, faced with a broken Web site, will simply stop visiting it rather than trying to sort out why it's broken. Of course, many Web sites instead employ a variety of technical tricks to get around ad blockers, ensuring their ads will still be displayed.

Given what's happening on the Web, it's reasonable to expect a similar fraction of smartphone users might want an ad blocker if it was available, with the concomitant arms race in ad block versus ad display technologies.

So long as users have not "rooted" their phones, a variety of core Android services can be relied upon by host applications to ensure that the ads they're trying to host are being properly displayed with the appropriate advertisement content. Similarly, advertising applications (or HTML ads) can make SSL connections to their remote servers, and even embed the remote server's public key certificate, to ensure they are downloading data from the proper source, rather than empty images from a transparent proxy.

Once a user has rooted their phone, of course, all bets are off. While it's hard to measure the total number of rooted Android phones, the CyanogenMod Android distribution, which requires a rooted phone for installation, is installed on roughly 722,000 phones—a tiny fraction of the hundreds of millions of Android phones reported to be in circulation. Given the relatively small market share where such hacks might be possible, advertisers might be willing to cede this fraction of the market rather than do battle against it.

Consequently, for the bulk of the smartphone marketplace, advertising apps on Android phones offer greater potential for blocking-detection and blocking-resistance than advertising on the Web, regardless of whether they are served by in-process libraries or by AdSplit. Given all the other benefits of AdSplit, we believe advertisers and application vendors would prefer AdSplit over the status quo.

Permissions and Privacy

Leaving aside whether it's legal for advertisers to collect sensitive information such as a user's precise location, we could always invent technical means to block this as a matter of policy. Unfortunately, a host app could always make its own requests, under its own authority, that violate the user's privacy and pass these into the AdSplit advertising app. Can we disincentivize such behavior? We hope that, if we can successfully reduce apps' default requests for privileges that they don't really need, then users will be less accustomed to seeing such permission requests. When they do occur, users will push back, refusing to install the app. (Reading through the user-authored comments in the Android Market, many apps with seemingly excessive permission requirements will have scathing comments, along with technical justifications posted by the app authors to explain why each permission is necessary.)

Furthermore, if advertisers ultimately prefer the AdSplit architecture, perhaps due to its improved resistance to click fraud and so forth, then they will be forced to make the tradeoff between whether they prefer improved integrity of their advertising platform, or whether they instead want less integrity but more privacy-violating user details.

Conclusion

AdSplit touches on a trend that will become increasingly prevalent over the next several years: the merger of the HTML security model and the smartphone application security model. Today, HTML is rapidly evolving from its one-size-fits-all security origins to allow additional permissions, such as access to location information, for specific pages that are granted those permissions by the user. HTML extensions are similarly granted varying permissions rather than having all-or-nothing access [6].

On the flip side, iOS apps originally ran with full, unrestricted access to the platform, subject only to vague policies enforced by human auditors. Only access to location information was restricted. In contrast, the Android security model restricts the permissions of apps, with many popular apps running without any optional permissions at all. Despite this, Android malware is a growing problem, particularly from third-party app stores (see, e.g., [4, 10]). Clearly, there's a need for more restrictive Android security, more like the one-size-fits-all Web security model.

While the details of how exactly Web apps and smartphone apps will eventually combine, our findings show where this merger is already underway: when Web content is embedded in a smartphone app. Well beyond advertising, a variety of smartphone apps take the strategy of using native code to set up one or more Web views and then do the rest in HTML and JavaScript. This has several advantages: it makes it easier to support an app across many different smartphone platforms. It also allows authors to quickly update their apps, without needing to go through a third-party review process.

These trends, plus the increasing functionality in HTML5, suggest that "native" apps may well be entirely supplanted by some sort of "mobile HTML" variant, not unlike HP/Palm's WebOS, where every app is built this way.

Maybe this will result in an industry battle royale, but it will also offer the ability to ask a variety of interesting security questions. For example, consider the proposed "Web intents" standard (http://webintents.org/). How can an "external" Web intent interact safely with the "internal" Android intent system? Both serve essentially the same purpose and use similar mechanisms. We, and others, will pursue these new technologies toward their (hopefully) interesting conclusions.

References

[1] T. Cheshire, "In Depth: How Rovio Made Angry Birds a Winner (and What's Next)," *Wired,* Mar. 2011: http://www.wired.co.uk/magazine/archive/2011/04/features/how-rovio-made-angry-birds-a-winner.

[2] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D.S. Wallach, "Quire: Lightweight Provenance for Smart Phone Operating Systems," 20th USENIX Security Symposium, San Francisco, CA, Aug. 2011.

[3] W. Enck, P. Gilbert, C. Byung-gon, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (OSDI '10), Oct. 2010, pp. 393–408.

[4] A.P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A Survey of Mobile Malware in the Wild," 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11), Chicago, IL, Oct. 2011.

[5] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe Exposure Analysis of Mobile In-App Advertisements," 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '12), Tucson, AZ, Apr. 2012.

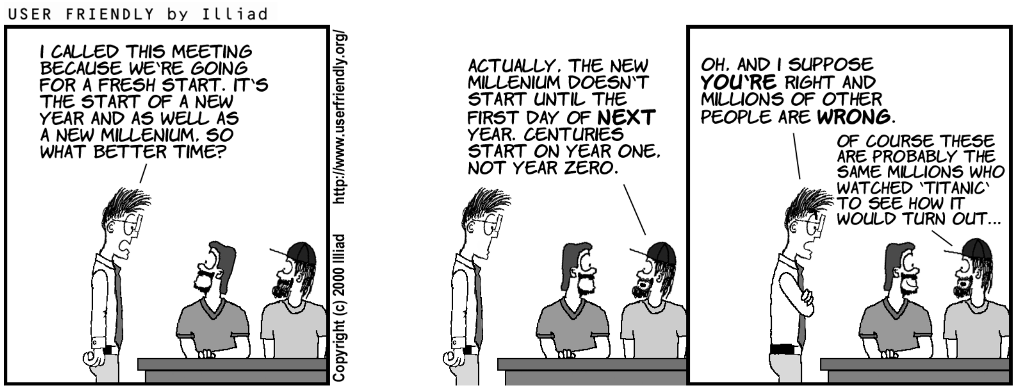
[6] L. Liu, X. Zhang, G. Yan, and S. Chen, "Chrome Extensions: Threat Analysis and Countermeasures," 19th Network and Distributed System Security Symposium (NDSS '12), San Diego, CA, Feb. 2012.

[7] L. McGann, "How Ars Technica's 'Experiment' With Ad-Blocking Readers Built on Its Community's Affection for the Site," Nieman Journalism Lab, Mar. 2010: http://www.niemanlab.org/2010/03/how-ars-technica-made-the-ask-of-ad-blocking-readers/.

[8] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: Zero-Day Protection for Smartphones Using the Cloud," Annual Computer Security Applications Conference (ACSAC '10), Austin, TX, Dec. 2010.

[9] S. Shekhar, M. Dietz, and D. Wallach, "Adsplit: Separating Smartphone Advertising from Applications," 21st USENIX Security Symposium, Bellevue, WA, Aug. 2012.

[10] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," 19th Network and Distributed System Security Symposium (NDSS '12), San Diego, CA, Feb. 2012.


USER FRIENDLY by Illiad
I CALLED THIS MEETING BECAUSE WE'RE GOING FOR A FRESH START. IT'S THE START OF A NEW YEAR AND AS WELL AS A NEW MILLENIUM. SO WHAT BETTER TIME?
http://www.userfriendly.org/
Copyright (c) 2000 Illiad
ACTUALLY, THE NEW MILLENIUM DOESN'T START UNTIL THE FIRST DAY OF NEXT YEAR. CENTURIES START ON YEAR ONE, NOT YEAR ZERO.
OH, AND I SUPPOSE YOU'RE RIGHT AND MILLIONS OF OTHER PEOPLE ARE WRONG.
OF COURSE THESE ARE PROBABLY THE SAME MILLIONS WHO WATCHED 'TITANIC' TO SEE HOW IT WOULD TURN OUT...

The Great Firewall of China
How It Blocks Tor and Why It Is Hard to Pinpoint

PHILIPP WINTER AND JEDIDIAH R. CRANDALL

Philipp Winter is a PhD student in the PriSec group at Karlstad University in Sweden. His work focuses on censorship resistance and analysis. Philipp would be happy if you would run a Tor relay if you are not operating one already.
philwint@kau.se

Jedidiah Crandall is an Assistant Professor at the University of New Mexico Department of Computer Science, where his research focuses on advanced inference techniques to understand the structure and operations of networks. He is also interested in natural language processing of Asian-language social media. He is typing this bio from behind the Great Firewall on a business trip, in Harbin where the dumplings and beer are excellent.
crandall@cs.unm.edu

Internet censorship is no longer a phenomenon limited to countries with a weak human rights record—the Western world is beginning to embrace the idea. This development leads to a fundamental research question: how can we know where the roadblocks are on the Internet and the details of how they work? In this article, we use the "Great Firewall of China" (GFC) to illustrate how complex of a problem it can be to find the network filtering devices, and how sophisticated the filtering itself can be when directed at an advanced target such as the Tor anonymity network.

The GFC is only a small part of the legal, regulatory, and technical mechanisms China has put in place for Internet censorship [6], but it is an important part because it helps to separate China's Internet from the Internet of the rest of the world. Without this, domestic control of Internet content would be moot because Chinese Internet users would simply seek out foreign Web sites where content was not controlled.

After all, the GFC is capable of much more than just filtering keywords. In this article, we will focus on two aspects.

We will first show the shortcomings in the current research literature that make it difficult to narrow down where Internet filtering occurs within China's Internet using Internet measurements.

In the second part of this article, we will show how the GFC is blocking the Tor anonymity network. Despite being originally designed as a low-latency anonymity network, Tor is increasingly used as a censorship circumvention tool.

Shortcomings in Our Understanding of China's Internet Censorship Implementation

What is censored is an important question to ask with respect to Internet censorship, but it is not directly related to the implementation of censorship and so will not be discussed in this section. There are two basic questions that should form the foundation for understanding any Internet censorship implementation in a given country: how is the filtering performed, and where does the filtering occur? In China, determining how filtering is performed is complicated because filtering implementations differ depending on location and, furthermore, can change over time. We discuss this issue in the following section and, with respect to the Tor network, later in this article. Where the filtering occurs is much more difficult to determine in China because China's Internet has a unique topology and may tunnel a large amount of IPv4 traffic through IPv6 tunnels.

How Is the Filtering Performed in China?

In 2003 Zittrain and Edelman [9] performed perhaps the first academic work to collect data about what China was filtering and how that filtering was implemented. Much of the work that followed in 2006 and 2007 focused on the filtering of HTTP GET requests based on keywords [2]. The data showed that for GET requests containing sensitive keywords (e.g., "falun" in reference to the Falun Gong religion), routers in China in between the client and server would forge multiple TCP RST segments in both directions to try to reset the TCP connection. In Clayton et al. [2], sequence number matching was used to determine that this filtering was probably being performed by a bank of intrusion detection systems (IDSes), where the packets are allowed to pass through for performance reasons but port mirrored to the bank of IDSes, which could use RST injection to stop the flow of traffic for connections in which keywords appeared. The ConceptDoppler project [3] also did some packet-level measurements of keyword filtering for a wider array of routes approximately one year after Clayton et al. The difference in time and place for these two sets of measurements may account for several differences in the results.

For example, one question that is now resolved but illustrates the difficulties in answering basic questions about Internet censorship in China is whether GET request filtering is stateful. That is, does any GET request packet with a blacklisted keyword trigger TCP RST segments, or should the TCP handshake be completed and an actual connection established first? Previous work (e.g., [2, 3]) had drawn different conclusions, but more recent work [8] found that the filtering is now totally stateful. This discrepancy in the literature is probably because some routes were stateful and some were not, but over time all routes upgraded to have stateful filters. This heterogeneity in the implementation of filtering in China and the fact that it is always evolving create many challenges for measuring the censorship implementation.

GET request filtering is relatively easy to measure because it appears to be symmetric and bi-directional so that eliciting censorship is as simple as sending an offending GET request to any IP address in China. In fact, the reader can trigger filtering by simply opening the URL http://www.baidu.com/s?wd=falun. China implements other types of filtering, including DNS, Web content filtering, and application-level filtering on microblog servers. The type of filtering most relevant to how China blocks Tor, which would be filtering by IP address, has been less well studied in the past literature.

Where Does the Filtering Occur in China?

The question of where a particular type of filtering occurs can be posed in different ways, but typically we are interested in whether filtering occurs on specific routes but not on others, and possibly what router on that route is performing the filtering.

Again, GET request filtering has been the most studied implementation of censorship-related filtering in China in this respect because it is easy to solicit filtering from outside the country. Clayton et al. [2] observed that the time-to-live (TTL) field of forged TCP RST packets was larger than that of packets that really came from the actual Web server on the other end of their connection. ConceptDoppler [3] manipulated the TTL field of packets with blacklisted keywords to locate which router on the route to each of the hosts within China they tested performed filtering and forged the RSTs. They concluded that the filtering was concentrated near the border but sometimes was as many as 13 hops beyond the border; 28% of the routes they tested had no filtering at all. Xu et al. [8] did a more comprehensive study and concluded that the filtering was occurring more at the provincial level.

In an article in the *Atlantic Monthly* in March 2008 [4], James Fallows wrote:

> In China, the Internet came with choke points built in. Even now, virtually all Internet contact between China and the rest of the world is routed through a very small number of fiber-optic cables that enter the country at one of three points: the Beijing-Qingdao-Tianjin area in the north, where cables come in from Japan; Shanghai on the central coast, where they also come from Japan; and Guangzhou in the south, where they come from Hong Kong. (A few places in China have Internet service via satellite, but that is both expensive and slow. Other lines run across Central Asia to Russia but carry little traffic.)

Xu et al.'s results and the fact that both GET request filtering and IP address filtering have at times been found not to occur on all routes into or out of China suggest a less centralized flow for China's international traffic. How does this square with Fallows' notion of a small number of choke points? The answer may lie in Internet Exchange Points (IXPs) and IPv4-over-IPv6 tunnels.

While the academic literature and online resources about IXPs seem only to refer to one IXP in Shanghai, the China Internet Network Information Center's (CNNIC) map of Internet connectivity in China (available at http://www1.cnnic.cn/images/info-stat/map1208.jpg) clearly shows three IXPs: in Beijing, Shanghai, and Guangzhou.

Why do these IXPs not appear in various efforts to locate IXPs on the Internet? The answer may lie in the fact that a large portion of China's Internet backbone appears to be implemented in IPv6, where IPv4 traffic is tunneled through in a "4-over-6" tunnel. "6-over-4" tunnels are more common and more well-studied than "4-over-6" tunnels, which along with IPv6 backbones create special challenges for any Internet measurement based on IPv4. Routing table information used in Internet topology measurements typically focuses on IPv4, and IPv4 traceroutes cannot detect hops inside an IPv6 tunnel because the TTL field will not be decremented in the IPv4 header. To measurements that are based on IPv4, "4-over-6" tunnels look like single hops. How much of China's Internet backbone is IPv6-based with IPv4 traffic being tunneled through? What percentage of international traffic traverses one of the three large IXPs in Beijing, Guangzhou, and Shanghai? The research literature does not have answers to these questions.

More Research Is Needed

Before we can begin to answer the question of where exactly Internet censorship filtering occurs within China's Internet, we need a better understanding of the structure of China's Internet. The roles of IPv6 and IXPs are key to this understanding. If a significant amount of China's backbone is IPv6, standard measurement techniques based on manipulating and observing IP TTLs will not allow us to find out which hop within this backbone is performing the filtering. It is possible that past efforts to locate the filtering, where the filtering has appeared to be near the border [3] or at the local provincial level [8] may have simply been seeing either the entrance point or exit point of a "4-over-6" tunnel because the results were based on IPv4 TTLs. This is compounded by other problems with using TTLs, such as the fact that forged RSTs from China appear to now make attempts to choose TTLs that appear to be from the other end of the connection.

In summary, more research is needed into both the structure of China's Internet in general and how to locate filtering specifically.

IP address blacklisting may take place at the same routers that implement GET request filtering, or it may be an entirely different mechanism. In either case, the structure of China's Internet will play a key role in the percentage of routes into or out of China that successfully block blacklisted IP addresses. Furthermore, it is likely that IP address blacklisting in China, like other mechanisms, is heterogeneous in the sense that various ISPs and different parts of the network may implement it differently (e.g., null routing, forged RSTs, network address translation, etc.).

The GFC and Tor

With roughly 400,000 daily users and 3,000 relays, Tor is the most popular low-latency anonymity network. Despite being originally designed for anonymity only, Tor turned out to be a good tool to circumvent censorship equipment and is now increasingly used for this purpose. This trend did not remain unnoticed by censors and is the reason why Tor is receiving special attention by the GFC, among others.

The Past

For several years, the Tor network has been in a cat-and-mouse game with the GFC. The first documented attempt to block Tor happened back in 2008. Users behind the GFC in China noticed that the official Tor Web site, www.torproject.org, stopped being reachable. As it turned out, deep packet inspection (DPI) boxes were scanning network traffic for the substring `torproject.org` in HTTP requests. When this substring was found, spoofed TCP reset segments were sent to both endpoints. Four years later (2012), this is still the case but can be circumvented easily by using HTTPS instead of plain HTTP. The DPI boxes are not able to detect the substring if the traffic is encrypted.

Although this type of block simply prevented users from downloading the Tor Browser Bundle from the official Web site (note that there are plenty of mirrors operated by volunteers), a user who somehow got her hands on a copy of the Tor client could still use the network without interference.

One year later, in 2009, the GFC's functionality was extended also to block all public relays as well as the directory authorities by simple IP blocks. The directory authorities serve the consensus, which is a directory containing all public Tor relays. The directory is downloaded by Tor clients during the bootstrapping phase, and blocking this step effectively blocked the public Tor network. But, at this point, the Tor developers had already implemented the concept of bridges, which are unpublished relays. Bridges are meant to provide a semi-hidden stepping-stone for censored users into the network. Along with bridges comes the bridge distribution problem: while in an ideal world, bridges should only be given to censored users, a censor can always mimic users and obtain—and then block—bridge addresses the same way. The current approach to the bridge distribution problem is to make it easy to get some of them but hard to get all of them, because then a censor could simply block them all. While the public network was blocked at this point in China, bridges remained functioning and were used heavily.

The increasing popularity of bridges did not remain unnoticed, though. Several months later, in March 2010, the Chinese bridge usage statistics started to drop significantly as shown on the right end in Figure 1. An explanation for this sudden drop was provided in a blog post: The GFC started to block some of the more popular bridges.

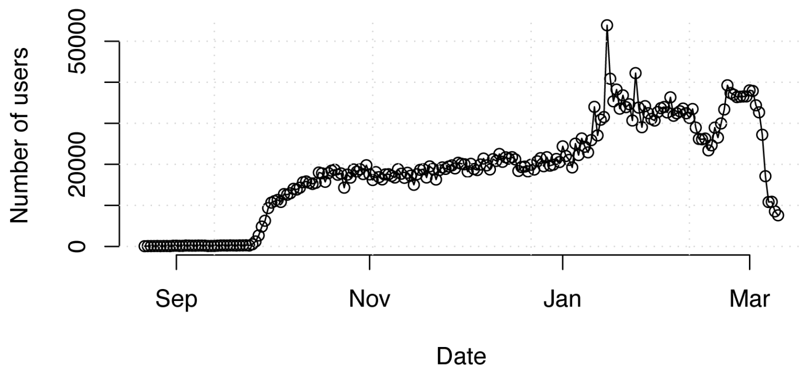


Figure 1: Bridge users connecting from China between 2009 and 2010

Bridges can be configured to be either public or private. A public bridge announces its existence to the public bridge database operated by the Tor developers so that it can be distributed automatically to people who need a bridge. A private bridge remains silent and hence only known to its operator.

At this point in the arms race it was still possible to set up private bridges and manually give their addresses to trusted people in China. For many months, private bridges remained a working, if less-than-ideal, solution to the unreachable public Tor network as well as to the mostly blocked public bridges; however, this changed in late 2011, when the GFC made the next move in the arms race. While the GFC's above-mentioned blocking attempts consisted mostly of simple IP blocks and Web site crawling, the next section outlines a drastic increase in sophistication and complexity.

The Present

In October 2011 a user reported on the Tor bug tracker that even private bridges appeared to get blocked within only minutes of their first use. As illustrated in Figure 2, the GFC is now using a novel two phase approach to make this possible [7]. In the first phase, Chinese egress traffic is being scanned for what appears to be Tor connections and the second phase is meant to confirm this suspicion by reconnecting to the suspected bridges and trying to initiate a Tor connection. The following two sections will present these two phases in greater detail.

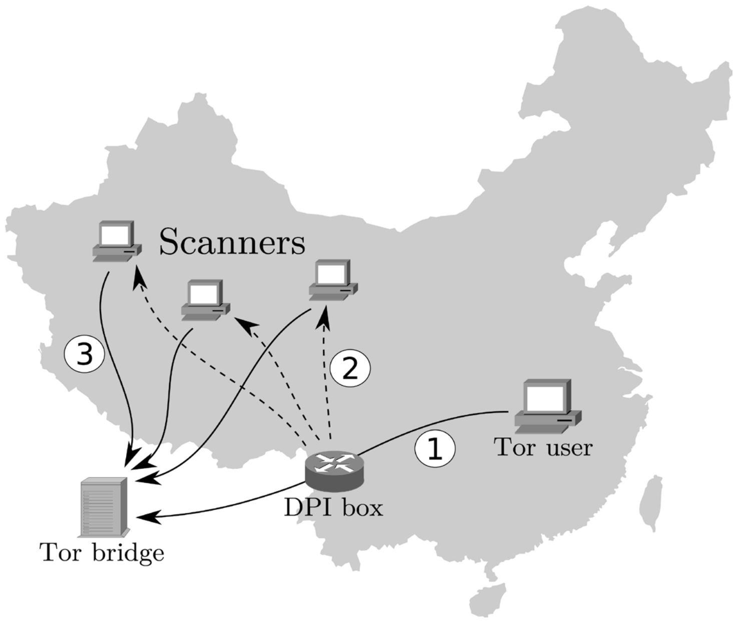


Figure 2: The GFC is (1) identifying Tor connections and (2) preparing scanners, which then (3) conduct follow-up scanning to verify that a Tor bridge was used

PHASE 1: FINGERPRINTING OF TOR

The GFC fingerprints Tor connections by exploiting the fact that Tor's TLS handshake slightly stands out from other applications' use of TLS. In particular, Chinese DPI boxes are looking for the cipher list, which is part of the TLS client hello. The cipher list is used by the Tor client to tell the bridge which cryptographic ciphers it supports. In particular, Tor's cipher list (for versions older than 0.2.3.17-beta) is a static string of 58 bytes. The following 58 bytes are what the DPI boxes are looking for in egress traffic:

```
0ac0 14c0 3900 3800 0fc0 05c0 3500 07c0 09c0 11c0
13c0 3300 3200 0cc0 0ec0 02c0 04c0 0400 0500 2f00
08c0 12c0 1600 1300 0dc0 03c0 fffe 0a00 ff00
```

For a long time, this cipher list was identical to the one used by Firefox 3. The Tor developers mimicked Firefox's cipher list in an attempt to make the Tor TLS handshake look like a Firefox connecting to an Apache Web server; however, newer versions of OpenSSL started adding `TLS_EMPTY_RENEGOTIATION_INFO_SCSV` to the cipher list, which made Tor's TLS handshake look different from Firefox 3 again.

PHASE 2: FOLLOW-UP SCANNING

Once Chinese DPI boxes discover a Tor-specific cipher list on the wire, follow-up scanning is initiated. Several minutes after the Tor handshake, two to three random Chinese IP addresses reconnect to the bridge just used by a Chinese user. These scanners connect to the bridge, start a TLS session, and then try to create a Tor circuit. If this succeeds, the bridge finds itself on the blacklist shortly thereafter. This blacklist consists of IP:Port tuples. Bridges and relays are not simply blacklisted by IP address. This might lead to overblocking, i.e., blocking more than actually necessary. Perhaps in an attempt to avoid this problem and minimize collateral damage, the GFC operators chose to block bridges by IP address and port. The blacklist probably holds around 4,000 such tuples (roughly 3,000 relays and 1,000 bridges).

The obvious step at this point would be for bridges to simply block these scanners. Unfortunately, the scanners mimic real computers very well. They come from almost random Chinese IP addresses, and the few properties in which they differ from real computers are hard to exploit effectively.

Evasion

Before discussing how the Tor network can be made more resistant to blocking attempts, we first describe the two fundamental ways in which Tor—and any other censorship evasion system—can be blocked:

BLOCK-BY-PROTOCOL

DPI boxes can be looking for Tor-specific signatures in the traffic exchanged by client and server. Ethiopia, for example, is blocking Tor by matching for signatures in the TLS client hello and server hello. If such a signature is found, the respective packet is simply dropped.

BLOCK-BY-ENDPOINT

Alternatively, Tor can be blocked by simply harvesting all relays and as many bridges as possible and blacklisting the respective IP addresses. Unfortunately, this is still a viable strategy.

EVADING PROTOCOL FILTERING

Protocol filtering can be evaded by obfuscating, scrambling, and reshaping a given network protocol to a degree that it is hard for DPI boxes to identify the target protocol. This can be done by simply exploiting "features" in TCP and IP or by adding a thin obfuscation layer between the transport and the application protocol. Both approaches can make the job of DPI boxes significantly harder.

The former is implemented by software such as fragroute (see http://www.monkey.org/~dugsong/fragroute/) or SniffJoke (see https://github.com/vecna/sniffjoke). Both projects exploit the fact that there is not enough information on the wire for a DPI box to fully reconstruct what is happening between two endpoints.

The latter is realized by a tool called obfsproxy, which is a network proxy developed by the Tor project. It is run as a local SOCKS proxy on the client-side as well as on the server-side. The actual obfuscation is handled by so-called pluggable transport modules. As long as the same module is loaded on both sides, the network traffic is being scrambled as dictated by the respective modules.

At this point, the only publicly available module for obfsproxy is called obfs2, which scrambles the network traffic so that no static fields remain that would be good candidates for fingerprinting. After a minimal handshake, the two parties have one symmetric session key each, which is used to build another layer of encryption over Tor's TLS connection.

EVADING ENDPOINT FILTERING

Assuming a perfect world in which Tor's transport protocol is unblockable, censors could still harvest and block the IP addresses of all bridges. After all, bridges are supposed to end up in the hands of legitimate users who need them, but not in the hands of evil censors; however, censors can always act as legitimate users to harvest addresses.

As already discussed above, the pragmatic approach so far has been to make it easy for a user to obtain a few bridges but hard to get many. Unfortunately, this approach is not very future-proof. Nation-state adversaries have lots of human resources, computational power, money, bandwidth, and IP address pools. Coming up with bridge distribution strategies that are robust against this kind of adversary is increasingly difficult.

Still, there is a glimpse of hope on the horizon. The recently proposed flash proxies concept by Fifield et al. [5] turns Web site visitors outside the censoring regime into short-lived stepping stones into the Tor network. The short-livedness is an advantage as well as a disadvantage. The disadvantage is that long-lived TCP connections can get terminated frequently. The advantage is that the mere volume of Web site visitors can be too much for a blacklist to handle. The censor should get overwhelmed by the sheer number of endpoints to block and discontinue blacklisting.

Conclusion

In this article, we gave an overview of the difficulties in determining in detail where filtering is taking place and how it is done. We used the Great Firewall of China and its ability to block the Tor anonymity network as an example.

Internet censorship is a relatively young field of research. Much more work needs to be done—both, in the field of measurement and circumvention—to keep the Internet free and information flowing freely. Current censorship research is exploring different areas. Blocking-resistant transport protocols that are being proposed appear to be pure randomness or mimic other protocols such as Skype or HTTP. Other research proposes to move circumvention to the Internet backbone or use Web site visitors as short-lived proxies. All in all, Internet censorship promises to be an exciting field of research with many important, challenging problems that will require bright minds to solve them.

References

[1] For a complete list of references, see: https://www.usenix.org/login-1212-winter-references.

[2] R. Clayton, S.J. Murdoch, and R.N.M. Watson, "Ignoring the Great Firewall of China," *I/S: A Journal of Law and Policy for the Information Society,* vol. 3, no. 2 (2007), pp. 70–77.

[3] J.R. Crandall, D. Zinn, M. Byrd, E. Barr, and R. East, "ConceptDoppler: A Weather Tracker for Internet Censorship," *Proceedings of the 14th ACM Conference on Computer and Communications Security* (ACM, 2007), pp. 352–365.

[4] J. Fallows, "The Connection Has Been Reset," *Atlantic Monthly,* March 2008: http://www.theatlantic.com/magazine/archive/2008/03/-the-connection-has-been-reset/306650/, accessed Sept. 6, 2012.

[5] D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, R. Dingledine, P. Porras, and D. Boneh, "Evading Censorship with Browser-Based Proxies," *Proceedings of the 12th Privacy Enhancing Technologies Symposium* (Springer, 2012), pp. 239–258.

[6] The Open Net Initiative, China: http://opennet.net/research/profiles/china, accessed Sept. 6, 2012.

[7] P. Winter and S. Lindskog, "How the Great Firewall of China Is Blocking Tor," *Proceedings of the 2nd Workshop on Free and Open Communications on the Internet* (Bellevue, WA, USA, 2012), USENIX Association.

[8] X. Xu, Z.M. Mao, and J.A. Halderman, "Internet Censorship in China: Where Does the Filtering Occur?" *Proceedings of the 12th International Conference on Passive and Active Measurement* (Springer, 2011), pp. 133–142.

[9] J. Zittrain and B. Edelman, "Internet Filtering in China," *IEEE Internet Computing,* vol. 7, no. 2, 2003, pp. 70–77.

Statement of Ownership, Management, and Circulation, 10/1/12

Title: ;login: Pub. No. 0008-334. Frequency: Bimonthly. Number of issues published annually: 6. Subscription price $125.

Office of publication: USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

Headquarters of General Business Office of Publisher: Same. Publisher: Same.

Editor: Rik Farrow; Managing Editor: Rikki Endsley, located at office of publication.

Owner: USENIX Association. Mailing address: As above.

Known bondholders, mortgagees, and other security holders owning or holding 1 percent or more of total amount of bonds, mortgages, or other securities: None.

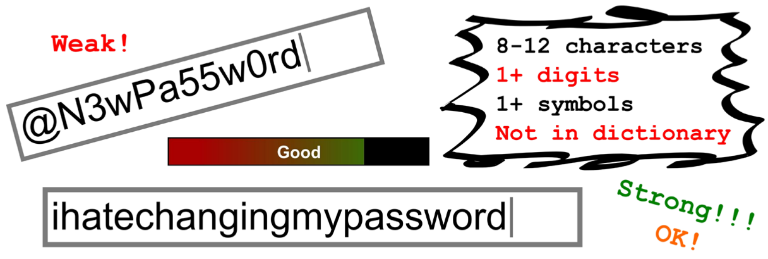
The purpose, function, and nonprofit status of this organization and the exempt status for federal income tax purposes have not changed during the preceding 12 months.

Extent and nature of circulation	*Average no. copies each issue issue during preceding 12 months*	*No. copies of single issue (Oct. 2012) published nearest to filing date*
A. Total number of copies	4063	3850
B. Paid circulation		
Outside-county mail subscriptions	2229	1858
In-county subscriptions	0	0
Other non-USPS paid distribution	1206	1051
Other classes	0	0
C. Total paid distribution	3435	2909
D. Free distribution by mail		
Outside-county	0	0
In-county	0	0
Other classes mailed through the USPS	66	51
E. Free distribution outside the mail	472	660
F. Total free distribution	538	711
G. Total distribution	3973	3620
H. Copies not distributed	90	230
I. Total	4063	3850
Percent Paid and/or Requested Circulation	86%	75%
Paid Electronic Copies	272	273
Total Paid Print Copies + Paid Electronic Copies	3707	3182
Total Print Distribution + Paid Electronic Copies	4245	3893
Percent Paid (Both Print and Electronic Copies)	87%	81%

I certify that the statements made by me above are correct and complete.
Casey Henderson, Co-Executive Director 10/1/12

Helping Users Create Better Passwords

BLASE UR, PATRICK GAGE KELLEY, SARANGA KOMANDURI, JOEL LEE, MICHAEL MAASS, MICHELLE L. MAZUREK, TIMOTHY PASSARO, RICHARD SHAY, TIMOTHY VIDAS, LUJO BAUER, NICOLAS CHRISTIN, LORRIE FAITH CRANOR, SERGE EGELMAN, AND JULIO LÓPEZ



Blase Ur is a second-year PhD student in the School of Computer Science at Carnegie Mellon University. His research focuses on usable security and privacy, including passwords, online behavioral advertising, and privacy decision making. He received his undergraduate degree in computer science from Harvard University. bur@cmu.edu

Patrick Gage Kelley is an Assistant Professor of Computer Science at the University of New Mexico. His research centers on information design, usability, and education around privacy. He recently completed his thesis at Carnegie Mellon University on standardized, user-friendly privacy displays for privacy policies and Android permission displays. pgage@cmu.edu

Saranga Komanduri is a PhD student in the School of Computer Science at Carnegie Mellon University. His research covers a broad spectrum of security-related topics, including authentication, usable security, and warnings. sarangak@cmu.edu

Over the past several years, we have researched how passwords are created, how they resist cracking, and how usable they are. In this article, we focus on recent work in which we tested various techniques that may encourage better password choices. What we found may surprise you.

Despite a litany of proposed password replacements, text-based passwords are not going to disappear anytime soon [4]. Passwords have a number of advantages over other authentication mechanisms. They are simple to implement, relatively straightforward to revoke or change, easy for users to understand, and allow for quick authentication; however, passwords also have a number of drawbacks. Foremost among these drawbacks is that it is difficult for users to create and remember passwords that are hard for an attacker to guess. Our research group at Carnegie Mellon University has been investigating strategies to guide users to create passwords that are both secure and memorable.

In particular, we have focused on techniques such as password-composition policies and password-strength meters—two of the most ubiquitous strategies employed by system administrators to help users create secure passwords. Although these strategies are commonly used, their effects had not been well understood. Through a series of online studies, we have aimed to understand how password-composition policies and meters affect password security, memorability, and user sentiment.

The first step in evaluating the security of a password is to understand the threat model. For instance, one can argue that a password that is hard to guess within the first three or five tries is secure, since an attacker would quickly be locked out. All but the most obvious passwords tend to resist this type of online attack. Passwords have been under attack in other ways due to a spate of password-database compromises in recent years, including at sites like Gawker and LinkedIn [3]. In possession of such a database, in which passwords are usually salted and hashed, rather than stored in plaintext, an adversary can still "crack" passwords by hashing potential passwords and checking whether these hashes appear in the database. This type of attack, known as an offline attack, is particularly pernicious since many users reuse a single password, or closely related passwords, across several sites to avoid remembering dozens of passwords [2]. Thus, an offline attack that successfully guesses a password on one site may let the attacker access a cornucopia of other accounts.

Joel Lee is an MS student in information security policy and management at Carnegie Mellon University. He is interested in usable security and privacy, adopting effective security policies in enterprises, and in balancing security with the core business operations of a company. He did his undergraduate degree in a partnership between CMU and Singapore Management University. jlee@cmu.edu

Michael Maass is a second-year PhD student studying software engineering at Carnegie Mellon University. He works on science of security problems, focused on sandboxing and evidence-based software assurance. Michael worked as a security engineer in the aerospace industry before pursuing a PhD. mmaass@cmu.edu

Michelle Mazurek is a fifth-year PhD student in electrical and computer engineering at Carnegie Mellon University. Her research focuses on usable security and privacy, including usable access control and passwords. She received her undergraduate degree in electrical engineering from the University of Maryland. mmazurek@cmu.edu

Timothy Passaro is a senior BS student in the Carnegie Institute of Technology and School of Computer Science at Carnegie Mellon University. His primary research interest is usable security and privacy. tpassaro@cmu.edu

Richard Shay is a fourth-year PhD student in the School of Computer Science at Carnegie Mellon University. His research focuses on usable privacy and security, studying online behavioral advertising and password policy. He received an undergraduate degree in computer science and classics from Brown University, and a master's degree in computer science from Purdue University. rshay@cmu.edu

In this article, we first introduce the methodology for our recent work on password-composition policies [5, 6] and password meters [8], and define the metrics we used to measure the security and usability of passwords. We then highlight key results from these studies, paying particular attention to the lessons they hold for guiding real-world password creation.

Methodology and Metrics

Both our password-composition-policy study and our password-meter study took place online in two separate parts. We recruited participants using Amazon's Mechanical Turk crowdsourcing service. Our password-composition-policy study involved more than 12,000 participants, while our study of password meters included more than 3,000 participants.

In the first part of each study, we asked participants to imagine that their main email provider had changed its password requirements, and that they needed to create a new password. In the study of password-composition policies, each participant created a password conforming to one of seven different composition policies, detailed later in this article. In the study of password meters, all participants created passwords under the same policy, but saw one of 14 different password meters, described later in this article, or no meter. Participants then completed a survey about the password-creation experience and were asked to re-enter their passwords. Two days later, participants received an email inviting them to return, log in again with their password, and to take another survey about how they handled their password.

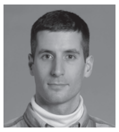
Traditionally, password strength for a set of passwords has been measured by entropy. In contrast, recent research advocates "guessability," the number of guesses it would take an adversary to guess a password, as a more appropriate metric for evaluating the real-world security of passwords against password-cracking attacks [1]. In our work, we calculated guessability by simulating a state-of-the-art password cracking algorithm [9] and determining how many attempts that algorithm would make to find a particular password, based on a particular set of training data.

To measure the usability of a password, we employed several metrics. First, we considered the memorability of the password. As a proxy for memorability, we examined the rate at which participants were able to log in successfully using their password about five minutes after password creation and when they returned for the second part of the study two or more days later. We also examined the rate at which participants returned for the second part of the study, hypothesizing that participants who created unmemorable passwords might not return. We further considered the proportion of participants who indicated in our surveys that they wrote their password down or stored it electronically, or who used their browser or a password manager to fill in their password automatically. Additionally, we presented participants with sentiment statements, to which they indicated levels of agreement or disagreement on a five-point Likert scale.

Password-Composition Policies

In our study of password-composition policies, we examined five main types of policies. Each participant was assigned round-robin to a single policy. As a baseline, the first policy, which we termed "basic8," required only that the password contain at least eight characters. To observe the impact of requiring longer passwords, we tested a "basic16" policy, which required only that the password

Timothy Vidas is an ECE PhD candidate at Carnegie Mellon University. His research interests include mobile device security, digital forensics, reverse engineering, cybercrime, and many other aspects of computer security. tvidas@cmu.edu


Lujo Bauer is an Assistant Research Professor in CyLab and the Electrical and Computer Engineering Department at Carnegie Mellon University. Lujo's research interests span many areas of computer security and include building usable access-control systems with sound theoretical underpinnings; developing languages and systems for runtime enforcement of security policies on programs; and, generally, narrowing the gap between a formal model and a practical, usable system. lbauer@cmu.edu

Nicolas Christin is the Associate Director of the Information Networking Institute at Carnegie Mellon University and a research faculty (Senior Systems Scientist) in CyLab and the Electrical and Computer Engineering and Engineering and Public Policy departments. His research is at the intersection of systems, security, and public policy, and has most recently focused on online crime, security economics, and psychological aspects of computer security. nicolasc@cmu.edu

Lorrie Faith Cranor is an Associate Professor of Computer Science and of Engineering and Public Policy at Carnegie Mellon University where she is Director of the CyLab Usable Privacy and Security Laboratory (CUPS). She is also a co-founder of Wombat Security Technologies, Inc. and previously was a researcher at AT&T Labs-Research. She has authored more than 100 research papers on online privacy, usable security, and other topics. lorrie@cmu.edu

contain at least 16 characters. We then tested a condition, "dictionary8," in which the password was stripped of non-alphabetic characters and checked against the free Openwall cracking dictionary. To test passwords that had to include several character classes, our "comprehensive8" condition mandated an eight-character password containing a lowercase letter, an uppercase letter, a digit, and a symbol. The password also needed to pass the same dictionary check as in dictionary8. We also tested three variants of a blacklist policy, which allowed all passwords containing at least eight characters other than passwords on blacklists, which were sourced from dictionaries ranging in size from hundreds of thousands to billions of potential passwords.

As shown in Figure 1, for weaker adversaries—those that would make around one billion guesses—the comprehensive8 and largest blacklist conditions were particularly resistant to a guessing attack, with the basic16 condition performing slightly worse. As the number of guesses increased, basic16 began to outperform the other conditions in guessing resistance. For instance, with one trillion guesses, only around half as many basic16 passwords were cracked as in comprehensive8 and the largest blacklist condition, which in turn were significantly more resistant to guessing than any other condition.

We also found usability advantages for the basic16 policy, which required long passwords with no further restrictions. Many popular Web sites' password policies are similar to our comprehensive8 condition, mandating passwords containing several character classes and passing a dictionary check; however, compared to participants who needed to comply with the comprehensive8 policy, those who needed to comply with basic16 needed fewer attempts to create their password and reused existing passwords at a lower rate [6]. Furthermore, basic16 participants expressed less frustration in our sentiment questions than those who needed to enter a password that does not appear in a dictionary or blacklist.

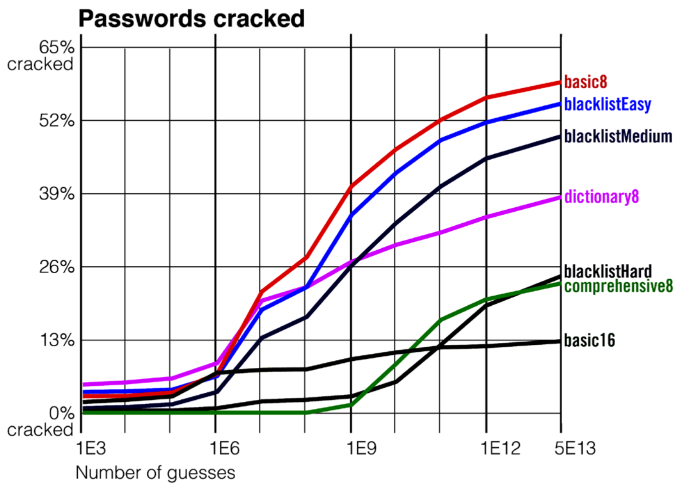


Figure 1: Percentage of passwords cracked by our password guesser for passwords collected under several password policies

Serge Egelman is a research scientist at UC Berkeley working on usable security problems. He uses empirical data to improve user interface designs for security mechanisms. He received his PhD from Carnegie Mellon University.
egelman@cs.berkeley.edu

Julio López is a Senior Software Engineer at Maginatics, Inc. where he works on large-scale cloud storage systems. He received his PhD and MS degrees from the Electrical and Computer Engineering Department at Carnegie Mellon University, and his BS from Universidad EAFIT in Colombia. julio.lopez@cmu.edu

Password-Strength Meters

In our study of password meters, participants were assigned round-robin to one of 15 conditions. In our control condition, participants were asked to create a password with no meter present. In each of the other 14 conditions, participants saw some variant of a password meter as they created their password. The design of one condition, which we termed a baseline meter, was informed by a survey we performed of password meter use on highly popular Web sites. Like the meters observed in the wild, our baseline meter computed the strength of the password using heuristics, such as the length of a password and the character classes it contained. To fill the bar completely, a participant's password could contain 16 or more characters, with no further restrictions. Alternatively, it could contain eight or more characters, including a lowercase letter, uppercase letter, digit, and symbol, as well as pass a dictionary check. As the bar became filled, it changed color from red to yellow to green; meanwhile, a single word of textual feedback changed in several steps from "bad" to "excellent." We also provided a suggestion for improvement, such as "Consider adding a digit or making your password longer."

Our other conditions, shown in Figure 2, tested meters with various visual elements and with different scoring strategies. To test the effect of the visual elements, we created seven meters that differed from the baseline meter only in their visual display. These conditions included a meter with a segmented, rather than continuous, bar; a meter that was always green; a tiny meter; a huge meter; a meter that didn't give suggestions for improving the password; and a meter that had only text, without a visual bar. We also created a meter that replaced the visual bar with an animated bunny. The stronger the participant's password, the faster the bunny danced.

To test the effect of changing how the meters scored passwords, we created four meters that scored passwords stringently, as well as two meters that nudged participants toward a particular password policy. Two of the four stringent meters had the same visual appearance as the baseline meter, yet always gave passwords half the score or one-third of the score that the baseline meter would have given. The two other stringent meters always gave half the score of the baseline meter, yet were text-only, lacking a visual bar. One text-only meter had standard-weight text, while the other had boldface text. One of the two meters nudging participants toward a particular policy only scored a password on its length, while the other policy more heavily weighted the inclusion of multiple character classes.

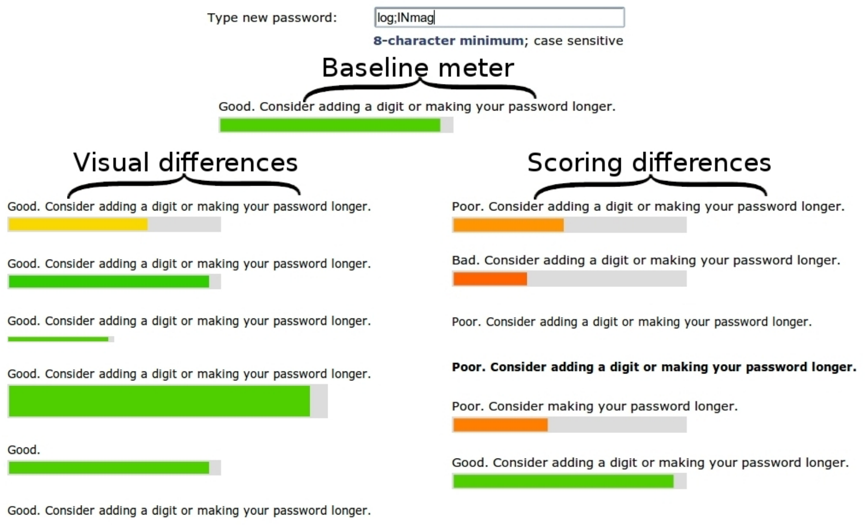
We found that all meters we tested led to passwords with different properties than those created without a meter. Passwords created with any type of meter were longer, on average, than those created with no meter. Furthermore, passwords created with stringent meters were the longest. For instance, passwords created with the half-score meter had a mean length that was 4.5 characters greater than those created with no meter.

We then evaluated the strength of passwords using the aforementioned "guessability" metric, quantifying the number of guesses a sophisticated adversary would need to guess that password. We found that all password meters we tested provided at least a small advantage against guessing attacks, although most of these differences were not statistically significant. As summarized in Table 1, the two stringent meters with visual bars, half and one-third score, provided a significant increase in guessing resistance compared to not having a meter. For instance, within the first 5 trillion guesses ($5{\times}10^{12}$), 47% of passwords created with no meter were cracked. In contrast, only 26% of passwords created with the half-score meter and 28% of passwords created with the one-third-score meter were cracked, while 34–46% of passwords created with all other meters were cracked.

	No Meter	Baseline Meter	Half-score Meter	One-third-score Meter	All Other Meters
5×10^{10} guesses	35%	27%	20%	17%	24–34%
5×10^{12} guesses	47%	39%	26%	28%	34–46%

Table 1: The percentage of passwords in each condition cracked within the first 5×10^{10} and first 5×10^{12} guesses

Although the passwords created with a meter tended to be longer and harder to guess, they did not seem to be less memorable. In particular, we did not observe statistically significant differences across conditions in any of our metrics for the memorability of passwords. Participant sentiment did differ across conditions, with the stringent meters leading participants to express annoyance at a higher rate. Stringent meters also caused increased participant disillusionment; participants in these conditions agreed at a higher rate that they did not "understand how the password strength meter rates [their] password" and agreed at a lower rate with the statement, "It's important to me that the password-strength meter gives my password a high score."



Type new password: log;INmag

8-character minimum; case sensitive

Good. Consider adding a digit or making your password longer.

Good. Consider adding a digit or making your password longer.

Good. Consider adding a digit or making your password longer.

Good. Consider adding a digit or making your password longer.

Good. Consider adding a digit or making your password longer.

Good.

Good. Consider adding a digit or making your password longer.

Poor. Consider adding a digit or making your password longer.

Bad. Consider adding a digit or making your password longer.

Poor. Consider adding a digit or making your password longer.

Poor. Consider adding a digit or making your password longer.

Poor. Consider making your password longer.

Good. Consider adding a digit or making your password longer.

Figure 2: The password meters we tested varied in their visual design and in the way they scored a password

Conclusions

From our study, we learned that a password-composition policy that causes users to create passwords that are longer than usual, rather than passwords that contain an array of character classes and that aren't in a blacklist, seems to possess a number of advantages. As the number of guesses increased, basic16 passwords were more resistant to a guessing attack. On the other hand, since long passwords are less common, it is also possible that existing cracking algorithms have not been optimized to crack long passwords. In addition to their greater resistance to an offline attack, basic16 passwords also had usability advantages over other policies resistant to guessing. For instance, these longer passwords were easier to create and led to more favorable participant sentiment.

Still, while encouraging users to make longer passwords seems to provide both security and usability benefits, we cannot yet definitively recommend a single password-composition policy as the ideal one. For instance, we are not yet sure of the optimal minimum length for these longer passwords. Likewise, a small percentage of the passwords created under the policy that emphasized length would have been guessed even by a very weak attacker. As such, we still need to establish whether encouraging or mandating the usage of additional character classes in long passwords would increase security without adversely affecting usability.

We began our password-meter study wondering whether meters had any effect, and we found that meters did indeed impact user behavior and security. For instance, meters led users to create longer passwords. Unfortunately, unless the meter scored passwords stringently, the resulting passwords were only marginally more resistant to password-cracking attacks. The non-stringent meters in our study most closely approximated real-world meters, suggesting that currently deployed meters are too lenient. This result hints that system administrators should make it more difficult to receive high scores from meters. How this result generalizes remains an open question since our study only examined the effect of a meter on the creation of a single password. It is possible a user would habituate to receiving lower scores from meters, neutralizing the security benefit. Also, whether having to remember multiple passwords would lead to greater difficulty in remembering passwords created with stringent meters is unknown.

As an alternative to guiding password creation, our group has also studied assigning users passphrases, which are long (and thus more secure) passwords formed of space-delimited words in a natural language. By automatically assigning passphrases, their guessability can be controlled. We compared the usability of these system-assigned passphrases to system-assigned traditional passwords. We found passphrases to be far from a usability panacea; by various usability metrics, they often performed slightly worse or no better than traditional passwords [7]. Our investigation also revealed a few ways in which passphrases may be improved to realize some of the desired usability benefits.

The research community is still far from providing the last word on the best strategies for helping users create passwords that are both hard to guess and easy to remember. Passwords have received substantial attention in recent years, with many exciting contributions coming from a number of different research groups. We look forward to conducting additional studies to help solidify our understanding of passwords, and to providing more definitive guidelines to help users create better passwords.

References

[1] J. Bonneau, "The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords," IEEE Symposium on Security and Privacy, 2012.

[2] D. Florencio and C. Herley, "A Large-Scale Study of Web Password Habits," WWW 2007.

[3] D. Goodin, "Why Passwords Have Never Been Weaker—And Crackers Have Never Been Stronger," Ars Technica, August 21, 2012.

[4] C. Herley, P. Van Oorschot, "A Research Agenda Acknowledging the Persistence of Passwords," *IEEE Security & Privacy Magazine,* vol. 10, no. 1, 2012, pp. 28–36.

[5] P.G. Kelley, S. Komanduri, M.L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L.F. Cranor, J. Lopez, "Guess Again (and Again and Again): Measuring Password Strength by Simulating Password-Cracking Algorithms," IEEE Symposium on Security and Privacy, 2012.

[6] S. Komanduri, R. Shay, P.G. Kelley, M.L. Mazurek, L. Bauer, N. Christin, L.F. Cranor, and S. Egelman, "Of Passwords and People: Measuring the Effect of Password-Composition Policies," CHI, 2011.

[7] R. Shay, P.G. Kelley, S. Komanduri, M. Mazurek, B. Ur, T. Vidas, L. Bauer, N. Christin, L.F. Cranor, "Correct Horse Battery Staple: Exploring the Usability of System-Assigned Passphrases," SOUPS, 2012.

[8] B. Ur, P.G. Kelley, S. Komanduri, J. Lee, M. Maass, M. Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer, N. Christin, and L.F. Cranor, "How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation," 21st USENIX Security Symposium, 2012.

[9] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek, "Password Cracking Using Probabilistic Context-Free Grammars," IEEE Symposium on Security and Privacy, 2009.

ReDeBug

Finding Unpatched Code Clones in Entire OS Distributions

JIYONG JANG, MAVERICK WOO, AND DAVID BRUMLEY

Jiyong Jang is a fifth-year PhD student in electrical and computer engineering at Carnegie Mellon University. His research interests include systems, software and network security, with a focus on malicious software analysis and binary program analysis. He received his BS degree in computer science and industrial system engineering in 2005 and his MS degree in computer science in 2007 from Yonsei University, South Korea. He is the recipient of the 2011 Symantec Research Labs Graduate Fellowship. jiyongj@cmu.edu

Maverick Woo is a postdoctoral researcher working in Carnegie Mellon CyLab. With a background in data structures and algorithm design, Maverick has a keen interest in applying insights from theoretical computer science to computer security. His recent work focuses on accelerating binary analysis through efficient abstraction recovery and automated reasoning. pooh@cmu.edu

David Brumley is an Assistant Professor at Carnegie Mellon University in the Electrical and Computer Engineering Department. Professor Brumley graduated from Carnegie Mellon University with a PhD in computer science in 2008. He has received several best paper awards, an NSF CAREER award, and the United Stated Presidential Early Career Award for Scientists and Engineers. dbrumley@cmu.edu

Programmers often copy code from one program to another. Unfortunately, when patches to buggy code are not propagated to all code clones, this leaves one or more programs still vulnerable. In this article, we present ReDeBug, a tool to identify unpatched code clones automatically that we have made available for use.

Unpatched Code Clones

Programmers should never fix the same bug twice. Unfortunately, buggy code often gets copied from project to project, and each project fixes the bug independently, which means resources are wasted to diagnose the same bug repeatedly. We call clones of buggy code that has been fixed in only a subset of projects *unpatched code clones*. Unpatched code clones are latent bugs that are likely to be vulnerable and can cause a serious vulnerability window—the time frame between when a vulnerability is disclosed and when a project containing the vulnerable code clone is fixed.

For example, the patch presented in Listing 1 was issued in July 2009 to fix a heap overflow bug in libvorbis. The patched vulnerability can cause a program crash or arbitrary code execution via a maliciously crafted OGG file [3]. Unfortunately, we found 93 unpatched code clones of this bug in our November 2011 data set. Projects including mplayer and libtritonus-java in Debian, mednafen and libvorbisidec in Ubuntu, and ffdshow and guliverkli in SourceForge all had the same vulnerable unpatched code. In this case, the 93 packages were exposed to this known vulnerability for more than 800 days past the initial patch date.

```
---   a/lib/res0.c
+++   b/lib/res0.c
@@    -208,10 +208,18 @@
      info->partitions=oggpack_read(opb,6)+1;
      info->groupbook=oggpack_read(opb,8);

+     /* check for premature EOP */
+     if(info->groupbook<0)goto errout;

+     for(j=0j;<info->partitions;j++){
          int cascade=oggpack_read(opb,3);
-         if(oggpack_read(opb,1))
-             cascade|=(oggpack_read(opb,5)<<3);
+         int cflag=oggpack_read(opb,1);
+         if(cflag<0) goto errout;
+         if(cflag){
+             int c=oggpack_read(opb,5);
+           if(c<0) goto errout;
+           cascade|=(c<<3);
+        }
         info->secondstages[j]=cascade;

         acc+=icount(cascade);
```

Listing 1: Example patch for CVE-2009-3379 showing diff details

To study how widespread the problem of unpatched code clone truly is and to provide a tool that can help developers fight against it, we developed ReDeBug [5], a system to find unpatched code clones quickly in code bases at the scale of *entire* OS distributions. Using ReDeBug, we examined more than 2.1 billion lines of code from all packages in Debian Lenny/Squeeze, Ubuntu Maverick/Oneiric, all C and C++ projects in SourceForge, and also the Linux kernel. ReDeBug identified 15,546 unpatched copies of known vulnerable code from 376 Debian/Ubuntu security-related patches. ReDeBug uses syntax-based pattern matching, which allows it to (1) scale to entire OS distributions, (2) support many different languages, and (3) guarantee zero false detections.

- **Scalability:** To give a sense of the scale necessary to find all unpatched code clones, observe that Debian Squeeze alone contains 16 GB of non-empty and non-comment code, spanning more than 348 million lines. Using ReDeBug on a machine with a 3.40 GHz i7 CPU and SSD, we were able to scan the 2.1 billion lines of code in our entire data set against 1,634 buggy code patterns in less than three hours. With the ability to search rapidly for unpatched code clones, ReDeBug can be used to improve the security of code bases in day-to-day development by promptly checking for copies of known vulnerabilities automatically.
- **Support for many different languages:** OS distributions include programs written in a variety of languages. For example, Debian Squeeze consists of 288 million lines of C/C++, 24 million lines of Java, 14 million lines of Python, 12 million lines of Perl, 5 million lines of PHP, and so on. To handle such a large variety of languages, ReDeBug uses a simple, fast, and language-agnostic syntax pattern matching approach to find unpatched code clones. We realize that there are more advanced matching algorithms that are applicable when the code is correctly parsed, and that such algorithms will likely find even more unpatched code clones. The challenge is, however, in the building of robust parsers for each language, which has proven difficult even for professional software assurance companies [1]. While we encourage future developers to add parsing support to ReDeBug, for now ReDeBug opts for a simpler robust algorithm that works across a wide variety of languages.
- **Zero false detection rate:** There are two types of false reports any clone detection algorithms can make. The first type is a syntactic "false detection." This happens when an algorithm says an unpatched code clone is present when it is not. ReDeBug eliminates false detections by performing a slower but exact match after all potential matches have been rapidly identified. In contrast, advanced heuristic matching algorithms used to find more code clones can suffer a higher false detection rate. Reporting only true matches to developers is important; otherwise, developers would end up wasting resources to examine the false reports. The second type is a semantic "false positive." This happens when an algorithm detects an unpatched code clone, but the clone is used in a non-vulnerable way, such as when checks have been inserted in earlier locations. Although ReDeBug inevitably can have false positives just like any other syntax-based method, we

argue that false positives still present problems because the code can be used in a vulnerable way due to a change in the future.

ReDeBug

ReDeBug is available for download as an open source tool on our Web site (http://security.ece.cmu.edu/redebug/). ReDeBug is written in Python to make the tool (1) easy to use without the need to compile first, (2) useful on multiple platforms, and (3) simple to extend with language-specific optimizations. The Web site also offers an online unpatched code clone detection service where developers can submit their code to test whether it contains known vulnerabilities stored in our database. If a match is found, a report showing both the original buggy code and unpatched code clones found in the submitted code is presented.

```
$ redebug.py -h
usage: redebug.py [-h] [-n NUM] [-c NUM] [-v] patch_path source_path

positional arguments:
  patch_path             path to patch files (in unified diff format)
  source_path            path to source files

optional arguments:
  -h, --help             show this help message and exit
  -n NUM, --ngram NUM    use n-gram of NUM lines (default: 4)
  -c NUM, --context NUM  print NUM lines of context (default: 10)
  -v, --verbose          enable verbose mode (default: False)
```

Listing 2: Help message of ReDeBug

A full technical description of ReDeBug has been presented in [5]. Here we concentrate on how to use ReDeBug to find unpatched code clones in practice. As shown in Listing 2, ReDeBug takes two positional arguments: patch path and source path. The first refers to the top-level patch directory from which we extract original buggy code snippets, and the second points to the top-level directory of the source tree to be checked. As optional arguments, -n defines how many lines of code are to be considered as a unit of code to compare, -c sets how many surrounding lines of code are to be reported as context, and -v enables verbose output. ReDeBug consists of three major components: (1) PatchLoader, which extracts original buggy code snippets from patch files; (2) SourceLoader, which matches source files against known buggy code; and (3) Reporter, which generates a report after performing exact-matching test. We explain each component of ReDeBug with an example of identifying the unpatched code clone for the CVE-2009-3379 vulnerability in the Debian mplayer package.

PatchLoader: ReDeBug takes patch files in the UNIX unified diff format, which is popular among open source developers. Listing 1 shows a patch for the CVE-2009-3379 vulnerability in libvorbis in the unified diff format. A unified diff patch consists of a sequence of diff hunks where each hunk includes the filename of a modified file, deleted source code lines that are prefixed by a -, and inserted source code lines that are prefixed by a +. Modifications are represented as deletions of old source code lines followed by insertions of new source code lines.

1. Consider a set of patches P_i. ReDeBug extracts original code snippets P_i' from P_i by excluding the lines prefixed by a "+" symbol. This is because the inserted lines are not present in original buggy code. The surrounding context lines are

included to conservatively identify unpatched code clones. ReDeBug requires only the patches but not the pre-patch source code. This allows ReDeBug to save significant space because we do not have to keep the original source code.

2. ReDeBug normalizes the extracted original buggy code P_i' to $\bar{P}_i$ by removing whitespaces except new lines and converting all characters into lowercase. We keep new lines since patches in the unified diff format operate at the line level. ReDeBug also identifies file types using the libmagic library, and performs language-specific normalization to increase the probability of identifying unpatched code clones. For example, for C, C++, and Java, we remove single line comments (//), multi-line comments (/* */), and curly brackets ({}). The code in Listing 3 shows the normalized buggy code extracted from the code in Listing 1. Regular expressions for such language-specific optimizations are defined in common.py, which can be easily extended to add more optimizations and support other languages.

```
info->partitions=oggpack_read(opb,6)+1;
info->groupbook=oggpack_read(opb,8);
for(j=0;j<info->partitions;j++)
int cascade=oggpack_read(opb,3);
if(oggpack_read(opb,1))
cascade|=(oggpack_read(opb,5)<<3);
info->secondstages[j]=cascade;
acc+=icount(cascade);
```

Listing 3: Normalized buggy code extracted from Listing 1

3. ReDeBug slides a window of n lines over the normalized code $\bar{P}_i$. For example, we have five windows from the code in Listing 3 when n=4: lines 1-4, 2-5, 3-6, 4-7, and 5-8. For each window w, we apply a list of hash functions H to build a list of hash values $h_i = \{ h(w) \mid w \in \bar{P}_i, h \in H \}$. At present, ReDeBug utilizes three hash functions: FNV-1a hash (http://isthe.com/chongo/tech/comp/fnv/), djb2 hash, and sdbm hash (http://www.cse.yorku.ca/~oz/hash.html) (refer to common.py). The default context in a diff file is three lines of code. Therefore, we can guarantee each window has at least one changed line by setting $n \geq 4$ (the default n is 4).

SourceLoader: ReDeBug builds a Bloom filter [2] for each source file to check the presence of known vulnerabilities. For example, ReDeBug checks for the CVE-2009-3379 vulnerability in the code in Listing 4 as follows:

```
info->begin=oggpack_read(opb,24);
info->end=oggpack_read(opb,24);
info->grouping=oggpack_read(opb,24)+1;
info->partitions=oggpack_read(opb,6)+1;
info->groupbook=oggpack_read(opb,8);

for(j=0;j<info->partitions;j++){
  int cascade=oggpack_read(opb,3);
  if(oggpack_read(opb,1))
    cascade|=(oggpack_read(opb,5)<<3);
  info->secondstages[j]=cascade;

  acc+=icount(cascade);
}
```

Listing 4: Source code snippet from mplayer package

```
info->begin=oggpack_read(opb,24);
info->end=oggpack_read(opb,24);
info->grouping=oggpack_read(opb,24)+1;
info->partitions=oggpack_read(opb,6)+1;
info->groupbook=oggpack_read(opb,8);
for(j=0;j<info->partitions;j++)
int cascade=oggpack_read(opb,3);
if(oggpack_read(opb,1))
cascade|=(oggpack_read(opb,5)<<3);
info->secondstages[j]=cascade;
acc+=icount(cascade);
```

Listing 5: Normalized source code snippet

1. ReDeBug normalizes source file F_j to $\bar{F}_j$ in a similar way by removing whitespaces except new lines and converting all characters into lowercase. Then, language-specific optimizations such as comment removal are applied according to the identified file type. For example, the code in Listing 4 is normalized into the code in Listing 5.
2. ReDeBug slides a window of *n*lines over the normalized source code $\bar{F}_j$. We hash each window *w* using the same list of hash functions *H*. Specifically, for each $h \in H$, we set the *h(w)*-th bit of the Bloom filter BF_j to 1. Each source file is now represented by its corresponding Bloom filter.
3. ReDeBug tests if a normalized source file $\bar{F}_j$ includes normalized buggy code $\bar{P}_i$ by checking if every bit in the locations specified by h_i is set to 1 in BF_j. For example, for all the hash values h_i generated from the code in Listing 3, we check if the corresponding bits are set to 1 in the Bloom filter built from Listing 5. If at least one of the bits is 0, that means the corresponding window of $\bar{P}_i$ is not present in $\bar{F}_j$. ReDeBug only records the pair
($\bar{P}_i$, $\bar{F}_j$) as a potential match if $\bar{F}_j$ contains the entire $\bar{P}_i$.

Reporter: For every pair ($\bar{P}_i$, $\bar{F}_j$) recorded, ReDeBug verifies whether $\bar{P}_i$ really occurs in $\bar{F}_j$. A Bloom filter may cause false detection due to hash collisions. This is why ReDeBug performs an exact match to eliminate any possible false detection due to the use of Bloom filters. For example, the code in Listing 5 indeed contains the buggy code in Listing 3. Finally, ReDeBug reports the Debian mplayer package contains an unpatched code clone of CVE-2009-3379. The report also presents a pair of the patch in Listing 1 and the matched source code in Listing 4, which helps developers to inspect the identified unpatched code clone easily.

Security-Related Bugs

Using ReDeBug, we analyzed more than 2.1 billion lines of source code from several OS distributions to comprehend the current trends of unpatched code clones. Table 1 shows the detailed breakdown of our collected source code data set. The Early 2011 dataset (Σ_1) consists of all source packages from Debian 5.0 Lenny, Ubuntu 10.10 Maverick, Linux Kernel 2.6.37.4, and all C/C++ projects in SourceForge. The Late 2011 dataset (Σ_2) contains all source packages from Debian 6.0 Squeeze and Ubuntu 11.10 Oneiric. For the SourceForge packages, we used version control systems such as Subversion, CVS, and Git to obtain up-to-date packages; then we excluded non-active code branches such as branches and tags directories. For source packages in Debian and Ubuntu, we applied existing patches, e.g., debian/patches/, because those patches can be included during a build. As a result,

	Distributions	Lines of Code	Date Collected
Early 2011 (Σ_1)	Debian Lenny	257,796,235	Jan 2011
	Ubuntu Maverick	245,237,215	Mar 2011
	Linux Kernel 2.6.37.4	8,968,871	Mar 2011
	SourceForge (C/C++)	922,424,743	Mar 2011
Late 2011 (Σ_2)	Debian Squeeze	348,754,939	Nov 2011
	Ubuntu Oneiric	397,399,865	Nov 2011
	Total	2,180,581,868	—

Table 1: Source code data set

Dataset	# Files	# Diffs	Date Released
Pre-2011 Patches (δ_1)	274	1,079	2001-2010
2011 Patches (δ_2)	102	555	2011
Total	376	1,634	—

Table 2: Security-related patch data set

the source packages we checked were patched with all available and included patches on the download date.

In order to find security-critical bugs, we collected security-related patches from Debian/Ubuntu security advisories that included the information about the corresponding packages and patches/diffs. We downloaded 376 security-related patches whose file names had recognizable CVE numbers, and gathered 1,634 diffs from these CVEs. As described in Table 2, pre-2011 patches (δ_1) were available at the time of collecting Σ_1, and 2011 patches (δ_2) were released between the download dates of Σ_1 and Σ_2.

In total, ReDeBug found 15,546 unpatched code clones in the two data sets Σ_1 and Σ_2. Figure 1 shows the detailed breakdown of unpatched code clones identified in Σ_1 and Σ_2 when querying for δ_1 and δ_2. We considered three scenarios to understand the current situation of unpatched code clones.

- $\{\delta_1 \& \delta_2\} \rightarrow \Sigma_1$: The unpatched code clones found in Σ_1 using δ_1 and δ_2 approximate how many (potentially) vulnerable packages an adversary may be able to spot when a patch becomes available. There were 10,248 unpatched code clones detected in the SourceForge data set. The old stable, but still supported on the

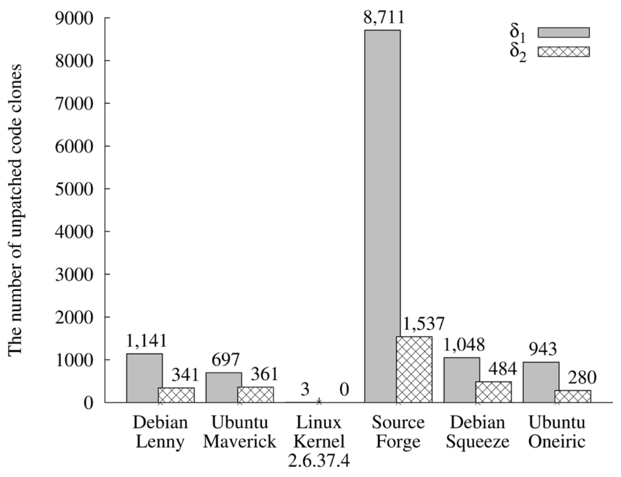


Figure 1: Unpatched code clones in Σ_1 and Σ_2

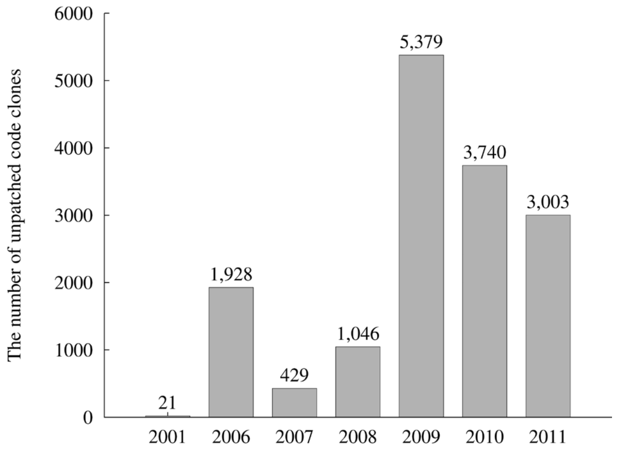


Figure 2: Unpatched code clones from patches in different years

download date, Debian Lenny and Ubuntu Maverick also had 1,482 and 1,058 unpatched code clones, respectively. When security-related bugs are fixed in the original packages, it is important to detect such serious vulnerabilities early before an adversary identifies them.

- $\{\delta_1 \& \delta_2\} \rightarrow \Sigma_2$: The unpatched code clones identified in Σ_2 using δ_1 and δ_2 roughly indicate how new versions of an OS respond to previously known security vulnerabilities. Debian Squeeze and Ubuntu Oneiric included 1,532 and 1,223 such unpatched code clones, respectively. We reported the 1,532 unpatched code clones identified in Debian Squeeze packages to the Debian security team and package developers. So far, 145 real bugs have been confirmed by developers either by private emails or by issuing a patch. This showcases the real world impact of ReDeBug. For some examples of the identified unpatched code clones, please refer to our paper [5] and our Web site, http://security.ece.cmu.edu/redebug/.
- $\delta_1 \rightarrow \Sigma_1$ vs. $\delta_1 \rightarrow \Sigma_2$: We investigated how many unpatched code clones persisted from the previous version of an OS to the latest version of an OS. In our evaluation, we compared the 1,838 unpatched code clones from δ_1 in Σ_1 and the 1,991 unpatched code clones also from δ_1 in Σ_2. Among these 3,829 clones, 1,379 persisted. Figure 2 shows the number of unpatched code clones identified from patches released in different years. Note that 21 of the unpatched code clones are security vulnerabilities that were patched over a decade ago (in 2001). This indicates that unpatched code clones are long-lived in modern OS distributions.

In some cases, unpatched code clones may be found in dead code, e.g., vulnerable code that is present but not included at build time or vulnerable code that is included but never gets executed due to logical conditions. The former usually happens when external library code is embedded in a source package, but the package is written to prefer the available system library to the embedded library. Dead code, however, may still be a latent vulnerability in that the accompanied vulnerable library code can be used depending on the availability of the system library during compilation on the user's machine. For C, specifically, we compile code with an assert statement inserted into the identified buggy code region and look for its corresponding assembly in the binary file to weed out such cases.

ReDeBug may have false positives when unpatched code clone is present but not vulnerable. For example, from the patch for CVE-2009-4016 shown in Figure 3a, an unpatched code clone was detected in ircd-ratbox package. The package maintainer informed us that the integer underflow vulnerability was fixed in a different location as shown in Figure 3b, which shows two new checks to guard against the vulnerable code. As a result, this unpatched code clone is used in a way that makes it unexploitable. ReDeBug and all other syntax-based approaches share the same problem.

Code Duplication

In order to understand the current situation of code clones, we performed a large-scale experiment to measure the overall amount of copied code in OS distributions. We measured this at two different granularities: the function level and the token (*n*-lines of source code) level.

First, for all C/C++ source files in the Debian Lenny code base, we roughly identified functions using the following Perl regular expression:

```
                else
                    *d++ = *src;
-               ++src;
-               --len;
    +           if (len > 0) {
    +               ++src, --len;
    +           }
            }
            *d = '\0';
            return dest;
```

3a: Patch for CVE-2009-4016

```
-       while (*src && (len > 0)) {
+       while (*src && (len > 1)) {
            if(*src & 0x80) {
                *d++ = '.';
                --len;
+               if(len <= 1)
    +               break;
            ...
            else
                *d++ = *src;
            ++src;
            --len;
        }
        *d = '\0';
        return dest;
```

3b: Another patch for CVE-2009-4016

Figure 3: Different fix for CVE-2009-4016

```
/^ \w+?\s[^;]*? \( [^;]*?\)\s*({ (?:[^{}]++|(?1))*})/xgsm
```

We realize that a regex may not be able to recognize all functions—that would require a complete parser; however, for our evaluation this is sufficient to provide an estimate of code duplication at the function level. We identified a total of 3,230,554 functions and measured their pairwise similarity using the Jaccard index. As shown in Figure 4, most of the function pairs had very low similarity (below 0.1), which is natural because different packages would have dissimilar code for different functionality; however, surprisingly, 694,883,223 pairs of functions had more than 0.5 similarity, and 172,360,750 of them were more than 90% similar. The result clearly shows a significant amount of code cloning, and this suggests that unpatched code clones will continue to be important and relevant in the future.

Second, we calculated the total fraction of shared tokens in each file for the SourceForge data set. As shown in Figure 5, about 30% of files were almost unique (0–10% shared tokens). In contrast, more than 50% of files shared more than 90%

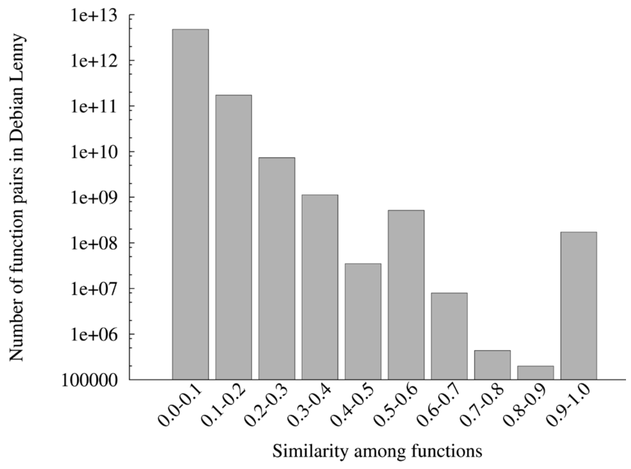


Figure 4: Similarity among functions

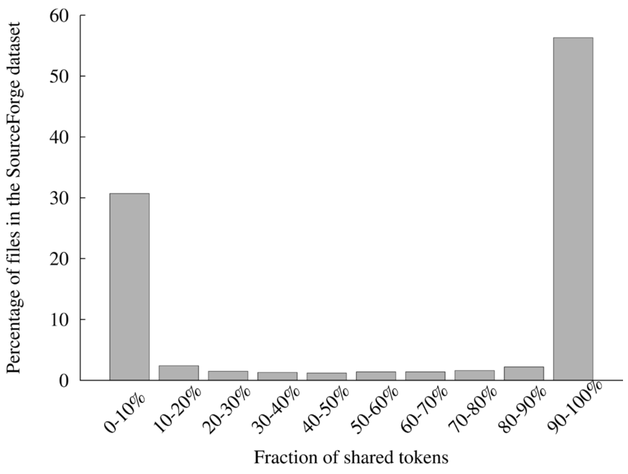


Figure 5: Fraction of shared tokens

of tokens with other files, which shows that code cloning is common within the SourceForge community as well. Note that 100% of shared tokens in a file does not necessarily mean it is copied from another file as a whole. For example, this could also happen when a file consists of small fractions from multiple files.

Related Work

Existing research has focused on finding all code clones, which is a harder problem than just identifying unpatched code clones. Finding all code clones potentially requires comparison among all code pairs, whereas identifying unpatched code clones can be done with a single sweep over the data set. This line of research uses a variety of matching heuristics based upon high-level code representations such as CFGs and parse trees. For example, CCFinder [7] generates a token sequence from a program using a lexer and transforms the token sequence based on language-dependent rules. A suffix-tree-based matching algorithm is then used to determine similar code. CP-Miner [8] parses a program, hashes its tokens into numeric values, and then runs the frequent subsequence mining algorithm to detect clone-related bugs. Deckard [6] and DejaVu [4] both build parse trees and represent structural information of a parse tree as a vector, and then cluster the vectors with respect to the Euclidean distance. An advanced heuristic matching, however, can suffer a higher false detection rate. For example, 73% of bug reports from CP-Miner and 37% of bug reports from DejaVu were false code clones. Furthermore, implementing good parsers is a difficult problem with which even professional software assurance companies struggle [1]. Of course, once that has been done, it will yield a robust level of abstraction that is not available to ReDeBug today.

Conclusion

We presented ReDeBug, a system to efficiently detect unpatched code clones. ReDeBug is designed to handle a large code base, e.g., an entire OS distribution written in a wealth of languages. We analyzed more than 2.1 billion lines of real code and identified 15,546 unpatched copies of known vulnerable code. This shows that the problem of unpatched code clone is persistent and recurring. The practical impact of ReDeBug has been confirmed by the 145 real bugs that were found and fixed in Debian Squeeze packages. We hope that ReDeBug can help developers enhance the security of their code in day-to-day development. You can try out ReDeBug by visiting our Web site at http://security.ece.cmu.edu/redebug/.

Acknowledgments

This research was supported in part by sub-award PO4100074797 from Lockheed Martin Corporation originating from DARPA Contract FA9750-10-C-0170 for BAA 10-36. This research was also supported in part by the National Science Foundation through TeraGrid resources provided by Pittsburgh Supercomputing Center. We would like to thank the anonymous referees of our related paper [5] and Debian developers for their feedback in this work.

References

[1] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler, "A Few

Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Communications of the ACM,* vol. 53, no. 2, 2010, pp. 66–75.

[2] Burton H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM,* vol. 13, no. 7, 1970, pp. 422–426.

[3] National Vulnerability Database, CVE-2009-3379: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3379, accessed 9/11/2012.

[4] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su, "Scalable and Systematic Detection of Buggy Inconsistencies in Source Code," Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, 2010.

[5] Jiyong Jang, Abeer Agrawal, and David Brumley, "ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions," Proceedings of the IEEE Symposium on Security and Privacy, 2012.

[6] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu, "Deckard: Scalable and Accurate Tree-Based Detection of Code Clones," Proceedings of the International Conference on Software Engineering, 2007.

[7] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Transactions on Software Engineering,* vol. 28, no. 7, 2002, pp. 654–670.

[8] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Transactions on Software Engineering,* vol. 32, 2006, pp. 176–192.

Practical Perl Tools

"Mala trom pee chock makacheesa."

DAVID BLANK-EDELMAN

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010. dnb@ccs.neu.edu

In this column, we hope to address the question "Is Perl the language of love?" And if that attempt falls short (as I suspect it will), is there a way to speak the language of love using Perl? Now, you and I may disagree which language is indeed the language of love. Is it Huttese, as in the title above (okay, really Quechua) or some other language? We're going to need some way to go back and forth between languages fluidly to answer this question.

One way to do this would be to use machine translation. Perhaps the most well known way to access this kind of translation is through a service Google provides called Google Translate. We're going to look at how to work with this service from Perl and also explore some of the little side lessons we can pick up along the way. There are three quick things I need to bring to your attention before we jump into how this all works:

1. Google Translate is not free to use; it used to be, back in the day, but now you have to pay a small amount to even play with it. Their pricing is listed on the Web site (https://developers.google.com/translate/v2/pricing). As of this writing, I'll be shelling out $20 US for the first one million characters of text being translated in this column. There is a separate charge of $20 US for using their language detection feature (again, per one million characters). To use the code found in this article or to write your own, you'll need to obtain your own Google Translate API key (https://code.google.com/apis/console/?api=translate) and also set it up so Google can bill you for the usage.
2. In addition to paying for Google Translate use, there are a whole bunch of other requirements you must adhere to in terms of how you need to identify its use in your application, branding, blah, blah, blah. Please read the "Attribution Requirements" and "HTML Markup Requirements" sections of their documentation (https://developers.google.com/translate/) carefully. You may wish to play the proper sections of John Williams' soundtrack in the background for proper effect.
3. There is a Perl module whose whole job is to hide the implementation details of using this service. I will indeed show you how to use it toward the end of this column, so if you are impatient, you can skip to the end. We're going to learn a bunch of cool things before we get there, but, hey, I'll understand if you are a busy bounty hunter and don't have the time to read all the way through.

Let's Take a REST

Many, many Web services these days provide some sort of REST API. REST stands for "Representational State Transfer." The Wikipedia article on REST at http://en.wikipedia.org/wiki/Representational_State_Transfer is decent (or was on the day I read it). Let me quote briefly from it:

> REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource
>
> . . . .
>
> Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: presented with a network of Web pages (a virtual state-machine), the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for his use.

(That last sentence comes from Roy Fielding's dissertation, which actually defined the REST architecture and changed Web services forever as a result.)

Discussing the REST idea in detail would get us into a whole other kettle of fish that might leave both of us smelling bad. For example, I think some people might quibble with Google calling their Google Translate API a REST API. They acknowledge this in their doc when they say "the Google Translate API is somewhat different from traditional REST. Instead of providing access to resources, the API provides access to a service."

They essentially bend the REST idea to say that using this service consists of constructing URLs with the right service request details in them, fetching the URL, and getting data back with the results of that service request. To quote their doc directly:

> "[T]he API provides a single URI that acts as the service endpoint.
>
> You access the Google Translate API service endpoint using the GET REST HTTP verb, as described in API operations. You pass in the details of all service requests as query parameters."

In this column, we've done this sort of thing lots of times. Let's use similar code to take a baby step:

```
  use LWP::Simple;

  my $tkey = '{YOUR Google Translate API KEY HERE}';

  my $results = get("https://www.googleapis.com/language/translate/v2/
languages?key=$tkey");

  print "$results\n";
```

This prints out what looks like a large data structure along the lines of:

```
{
 "data": {
  "languages": [
   {
    "language": "af"
   },
   {
    "language": "ar"
   },
   ...
    {
    "language": "de"
   },
   {
    "language": "el"
   },
   {
    "language": "en"
   },
   ...
   {
    "language": "vi"
   },
   {
    "language": "yi"
   },
   {
    "language": "zh"
   },
   {
    "language": "zh-TW"
   }
   ]
  }
}
```

And with that, we've made our first Google Translate API call. In this case we've asked for the list of languages supported by the service.

When you look at the output, you may be thinking, "Hey, that output looks like a data structure but is pretty legible; what format is it in?" Glad you asked. The Google Translate API returns data in JSON (JavaScript Object Notation) format. JSON has become one of the lingua francas of Web services on the Net. We've seen JSON in this column before because it is a kissing cousin (where kissing cousin might be better described as "a subset") of the YAML data serialization format that shows up all around the Perl world. Given JSON's ubiquity, knowing how to work with it is a desirable skill that is easy to pick up.

As an example, we could modify our previous code to read like this:

```
    use LWP::Simple;
    use Data::Dumper;
    use JSON;

    my $tkey = '{YOUR Google Translate API KEY HERE}';

    my $results = get(
       "https://www.googleapis.com/language/translate/v2/
languages?key=$tkey");

    my $decoded = decode_json $results;

    foreach my $language ( @{ $decoded->{data}->{languages} } ) {
       print values $language, "\n";
    }
```

and it would print out the list of supported languages. Let's take apart the new parts of this code so it is clear. We've loaded the JSON module that turns JSON data into a Perl data structure. In this case, it has returned a hash reference to a data structure that looks like this (courtesy of the Perl debugger):

```
    -> HASH(0x7f88f4499e48)
        'data' => HASH(0x7f88f44eef68)
          'languages' => ARRAY(0x7f88f4760ad8)
            0  HASH(0x7f88f462efb0)
              'language' => 'af'
            1  HASH(0x7f88f462ef98)
              'language' => 'ar'
     ...
```

There's an outer hash with the single key of `'data'` that contains a reference to an inner hash whose only key is `'languages'` and whose value is a reference to a Perl array. Each element of that array contains a reference to a hash with `'language'` as its key and the name of the language as the value.

This line:

```
     $decoded->{data}->{languages}
```

returns the reference to the languages array. We dereference it (using the @{something} notation) to get at the values in the array, iterate over them, and print the list of values contained in the hash stored in each value.

If your first reaction to this code is "yucko," I don't blame you. It sure seems like a lot of work to get a single list of languages. But if you look carefully at the JSON excerpt that is coming back from the service as printed above, you'll see that the JSON decode() call is faithfully translating the JSON structure into the correct Perl data structure. In the real world you would encapsulate this call into a routine in your code that would construct a more pleasant data structure for the rest of the code to share. C'est la vie.

Translate Something Already!

Let's actually get to the translation process. To do so we have to add a few more twists to our previous code:

1. We have to make sure we're playing by the rules of the (Web) road. Anything we send to Google Translate in a URI must be properly escaped. Luckily, we have a few Perl modules available to us that make this easy.
2. Depending on what sort of code you are writing, you may have to deal with going back and forth between different character encodings (UTF-8, Latin-1, etc.). Character encoding is a serious rabbit hole in itself, so we're not going to talk much about it for fear of being sucked into an entirely separate column. I did sneak one character encoding decision in the previous code. The routine we used from the JSON module, decode_json(), expects to receive a UTF-8 encoded string and interpret it as UTF-8 text. You'll see a similar, albeit more overt, decision in the code we're about to construct.
3. As much as I appreciate the simplicity of LWP::Simple, we are shortly going to be at the point where we are going to have to assert more control over just how the request is constructed and sent out. For example, if the text that you are translating is over a certain length (Google's docs says 5000 characters, but I've seen some indication that 2000 characters might be a safer limit), the request must be sent as a POST instead of the usual GET type. And when you do this, Google Translate requires you also send along a header that says "X-HTTP-Method-Override: GET" (i.e., treat this POST request as if it were a GET). I'm not going to show code that takes this limit into account because it is built into one of the modules we'll see later, but I just wanted to let you know about it should you start writing something on your own and find LWP::Simple can't get this fancy. We'll use its big brother, LWP::Agent, in the examples below, but there are a number of possible modules we could use.

A quick aside that originates from #3 above: in the process of researching this article, I came upon a module I had never seen before called REST::Client. REST::Client describes itself as "A simple client for interacting with RESTful http/https resources." It basically provides a little bit of syntactic sugar that makes the fairly simple task of talking to a REST server even simpler. That's cool, but even cooler is the module based on it called REST::Client::Simple. REST::Client::Simple lets you describe the API you are communicating with (e.g., resources you can access, parameters that can be passed) and then write code that speaks in terms of that API. For example, excerpted from the documentation:

```
 use Net::CloudProvider;

my $nc = Net::CloudProvider(user => 'foobar', api_key => 'secret');
my $response = $nc->create_node({
    id                              => 'funnybox',
    hostname                        => 'node.funnybox.com',
    os                              => 'debian',
    cpus                            => 2,
    memory                          => 256,
    disk_size                       => 5,
 });
```

This code started off with a largish definition (not printed here) that said there exists a create_node command performed by making a POST call with certain possible parameters for that resource. Note that the code above doesn't show anything about how the actual POST request is constructed or sent: that's all done in the background by REST::Client::Simple. This level of sophistication is beyond what we need for our translation example code, but I thought you might find this module handy some day.

Okay, so let's actually translate some stuff. To do so, we'll need to construct a URI with the following parameters:

- key: we saw this before—it is our API key
- q: the string we want to translate
- target: the target language for the translation
- source: the source language, if we don't want Google Translate to try and guess it

Here's a small program that takes an English string as an argument and prints Google Translate's best guess at its French equivalent:

```
use LWP::UserAgent;
use URI::Escape;
use JSON;

my $tkey = '{YOUR Google Translate API KEY HERE}';

my $sl = 'en';
my $tl = 'fr';

my $query = URI::Escape::uri_escape_utf8( $ARGV[0] );

my $ua = LWP::UserAgent->new();

my $results
   = $ua->get( 'https://www.googleapis.com/language/translate/v2/'
      . "?key=$tkey"
      . "&q=$query"
      . "&source=$sl"
      . "&target=$tl" );

die "Translation failed: " . $results->status_line
   unless $results->is_success;

my $decoded = decode_json $results->decoded_content;

binmode(STDOUT, ":utf8");

foreach my $translation ( @{ $decoded->{data}->{translations} } ) {
   print values $translation, "\n";
}
```

This code isn't very different from the previous examples. The code uses a slightly different URI that requests translations, and it adds a few small things to make sure that we stay URI and UTF-8 legal (URI::Escape::uri_escape_utf8 to encode the URI before sending, binmode(STDOUT, ":utf8") to make sure our output to STDOUT is kept UTF-8).

The process is still pretty simple. Let's see how to make it even simpler using a custom module for the job.

WWW::Google::Translate and Beyond

There's a lovely module called WWW::Google::Translate that makes writing code for this API even easier than what we saw above. The module takes into account the encoding, JSON, text size, and other details so you don't have to. It can do spiffy things such as cache the results so future requests for the same text don't force you to use up more of your API calls.

Here's the first example from the docs:

```
use WWW::Google::Translate;

my $wgt = WWW::Google::Translate->new(
    {   key            => '<Your API key here>',
        default_source => 'en',   # optional
        default_target => 'ja',   # optional
    }
);
my  $r = $wgt->translate( { q => 'My hovercraft is full of eels' } );
for my $trans_rh (@{ $r->{data}->{translations} }) {
    print $trans_rh->{translatedText}, "\n";
}
```

Given our previous code, you can probably guess we just need to create a new WWW::Google::Translate object and then ask the module to call a translate() method to get the job done. One quick note about this example: if you run this code using later versions of Perl, you may get a non-fatal error that looks something like this:

```
Wide character in print at Desktop/untitled.pl line 16.
```

This is warning you that you were trying to print Unicode data without preparing STDOUT to receive them. The fix for this is to add the line from above to the program before the print takes place:

```
binmode(STDOUT, ":utf8");
```

Okay, one last addition before we draw this column to a close. If you want to write code that performs translations but isn't directly tied to Google Translate, you may wish to check out the Lingua::Translate set of modules. Lingua::Translate is the closest equivalent to the DBI framework for databases.

With Lingua::Translate, you write the same code independent of the back-end service. Lingua::Translate::Google describes itself as "mostly a wrapper for the WWW::Google::Translate module," so you basically can use the power of WWW::Google::Translate while keeping your code back-end neutral. If later on you decide to stop using Google Translate and switch to something else, very little will have to change in the rest of your program.

Take care and I'll see you next time.

Data Processing with Pandas

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

In most of my past work, I've always had a need to solve various sorts of data analysis problems. Prior to discovering Python, AWK and other assorted UNIX commands were my tools of choice. These days, I'll mostly just code up a simple Python script (e.g., see the June 2012 *;login:* article on using the collections module). Lately though, I've been watching the growth of the Pandas library with considerable interest.

Pandas, the Python Data Analysis Library, is the amazing brainchild of Wes McKinney (who is also the author of O'Reilly's *Python for Data Analysis*). In short, Pandas might just change the way you work with data. Introducing all of Pandas in a short article is impossible here, but I thought I would give a few examples to motivate why you might want to look at it.

Preliminaries

To start using Pandas, you first need to make sure you've installed NumPy (http://numpy.scipy.org). If you've primarily been using Python for systems programming tasks, you may not have encountered NumPy; however, it gives Python a useful array object that serves as the cornerstone for most of Python's science and engineering modules (including Pandas). Unlike lists, arrays can only consist of a homogeneous type (integers, floats, etc.). Operations involving arrays also tend to operate on all of the elements at once. Here is a short example that illustrates some differences between lists and arrays:

```
>>> # Python lists
>>> c = [1,2,3,4]
>>> c * 3
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
>>> c + [10,11,12,13]
[1, 2, 3, 4, 10, 11, 12, 13]
>>> import math
>>> [math.sqrt(x) for x in c]
[1.0, 1.4142135623730951, 1.7320508075688772, 2.0]
>>>

>>> # numpy arrays
>>> import numpy
>>> d = numpy.array([1,2,3,4])
>>> d * 3
```

```
array([ 3,  6,  9, 12])
>>> d + numpy.array([10,11,12,13])
array([11, 13, 15, 17])
>>> numpy.sqrt(d)
array([ 1.        ,  1.41421356,  1.73205081,  2.        ])
>>>
```

Once you've verified that you have NumPy installed, go to the Pandas Web site (http://pandas.pydata.org) to get the code before trying the examples that follow.

Analyzing CSV Data

One of my favorite pastimes these days is to play around with public data sets. Another one of my favorite activities has been riding around on my road bike—something that was recently curtailed after I hit a huge pothole and had to have my local bike shop build a new wheel. So, in the spirit of huge potholes, let's download the city of Chicago's pothole database from the data portal at http://data.cityofchicago.org. We'll save it to a local CSV file so that we can play around with it.

```
>>> u = urllib.urlopen("https://data.cityofchicago.org/api/views/7as2-ds3y/
rows.csv")
>>> data = u.read()
>>> len(data)
27683443
>>> f = open('potholes.csv','w')
>>> f.write(data)
>>> f.close()
>>>
```

As you can see, we now have about 27 MB of pothole data. Here's a sample of what the file looks like:

```
>>> f = open('potholes.csv')
>>> next(f)
'CREATION DATE,STATUS,COMPLETION DATE,SERVICE REQUEST NUMBER,TYPE OF
SERVICE REQUEST,CURRENT ACTIVITY,MOST RECENT ACTION,NUMBER OF POTHOLES
FILLED ON BLOCK,STREET ADDRESS,ZIP,X COORDINATE,
Y COORDINATE,Ward,Police District,Community Area,LATITUDE,LONGITUDE,LOCATI
ON\n'
>>> next(f)
'09/20/2012,Completed - Dup,09/20/2012,12-01644985,Pot Hole in Street,,,0,
172 W COURT PL,60602,1174935.20259427,1901041.84281984,42,1,32,
41.883847164125186,-87.63307578849374,"(41.883847164125186,
-87.63307578849374)"\n'
>>>
```

Pandas makes it extremely easy to read CSV files. Let's use its read_csv() function to grab the data:

```
>>> import pandas
>>> potholes = pandas.read_csv('potholes.csv', skip_footer=True)
>>> potholes
<class 'pandas.core.frame.DataFrame'>
Int64Index: 116718 entries, 0 to 116717
Data columns:
```

```
CREATION DATE                       116718  non-null values
STATUS                              116718  non-null values
COMPLETION DATE                     115897  non-null values
SERVICE REQUEST NUMBER              116718  non-null values
TYPE OF SERVICE REQUEST             116718  non-null values
CURRENT ACTIVITY                    94429  non-null values
MOST RECENT ACTION                  94191  non-null values
NUMBER OF POTHOLES FILLED ON BLOCK  93790  non-null values
STREET ADDRESS                      116717  non-null values
ZIP                                 115705  non-null values
X COORDINATE                        116660  non-null values
Y COORDINATE                        116660  non-null values
Ward                                116695  non-null values
Police District                     116695  non-null values
Community Area                      116696  non-null values
LATITUDE                            116660  non-null values
LONGITUDE                           116660  non-null values
LOCATION                            116660  non-null values
dtypes: float64(9), object(9)
>>>
```

When reading data, Pandas creates what's known as a DataFrame object. One way to view a DataFrame is as a collection of columns. In fact, you can easily extract specific columns or change the data:

```
>>> addresses = potholes['STREET ADDRESS']
>>> addresses[0:5]
0   172 W COURT PL
1   1413 W 17TH ST
2   11800 S VINCENNES AVE
3   3499 S KEDZIE AVE
4   1930 W CULLERTON ST
Name: STREET ADDRESS
>>> addresses[1] = '5412 N CLARK ST'
>>>
```

And there is so much more that you can do. For example, if you wanted to find the five most reported addresses for potholes, you could use this one-line statement:

```
>>> potholes['STREET ADDRESS'].value_counts()[:5]
4700 S LAKE PARK AVE    108
1600 N ELSTON AVE       84
7100 S PULASKI RD       80
1000 N LAKE SHORE DR    80
8300 S VINCENNES AVE    73
>>>
```

Let's say you want to find all of the unique values for a column. Here's how you do that:

```
>>> # Get possible values for the 'STATUS' field
>>> potholes['STATUS'].unique()
array([Completed - Dup, Completed, Open - Dup, Open], dtype=object)
>>>
```

Here is an example of filtering the data based on values for one of the columns:

```
>>> fixed = potholes[potholes['STATUS'] == 'Completed']
>>> fixed
<class 'pandas.core.frame.DataFrame'>
Int64Index: 94490 entries, 1 to 116717
Data columns:
CREATION DATE       94490  non-null values
STATUS              94490  non-null values
...
>>>
```

In this example, the relation `potholes['STATUS'] == 'Completed'` is computed across all 116,000 records at once and creates an array of Booleans. By using that array as an index into potholes, we get only those records that matched as True. It's kind of a neat trick.

In addition to street addresses, the pothole data also includes the total number of potholes fixed at each address. Let's try to refine our analysis so that it takes this into account. Specifically, we'd like to sum up the total number of potholes fixed at each address and base our report on that. Here's how to do it.

First, let's just pick out data on street addresses and number of potholes:

```
>>> addr_and_holes = fixed[['STREET ADDRESS',
...                   'NUMBER OF POTHOLES FILLED ON BLOCK']]
>>> addr_and_holes
<class 'pandas.core.frame.DataFrame'>
Int64Index: 94490 entries, 1 to 116717
Data columns:
STREET ADDRESS                        94489  non-null values
NUMBER OF POTHOLES FILLED ON BLOCK    93558  non-null values
dtypes: float64(1), object(1)
>>>
```

Next, let's drop missing values in the data:

```
>>> addr_and_holes = addr_and_holes.dropna()
>>> addr_and_holes
<class 'pandas.core.frame.DataFrame'>
Int64Index: 93558 entries, 13 to 116717
Data columns:
STREET ADDRESS                        93558  non-null values
NUMBER OF POTHOLES FILLED ON BLOCK    93558  non-null values
dtypes: float64(1), object(1)
>>>
```

Let's group the data by street address and calculate totals:

```
>>> addr_and_totals = addr_and_holes.groupby('STREET ADDRESS').sum()
>>> addr_and_totals[:5]
            NUMBER OF POTHOLES FILLED ON BLOCK
STREET ADDRESS
1 E 100TH PL        9
1 E 110TH PL       20
1 E 111TH ST       10
```

```
1 E 11TH ST           20
1 E 121ST ST          21
>>>
```

Finally, let's sort the results:

```
>>> addr_and_totals = addr_and_totals.sort('NUMBER OF POTHOLES FILLED ON
BLOCK')
>>> addr_and_totals[-5:]
                  NUMBER OF POTHOLES FILLED ON BLOCK
STREET ADDRESS
6300 N RAVENSWOOD AVE     461
8200 S MARYLAND AVE       498
3900 S ASHLAND AVE        575
12900 S AVENUE O          577
5600 S WOOD ST            664
>>>
```

And there you have it—the five worst blocks on which to ride your road bike. It's left as an exercise to the reader to take this data and extend it to find the worst overall street on which to ride your bike (by my calculation it's Ashland Avenue, which is probably of no surprise to Chicago residents).

A File System Example

Let's try an example involving a file system. Define the following function that collects information about files into a list of dictionaries:

```
import os

def summarize_files(topdir):
   filedata = []
   for path, dirs, files in os.walk(topdir):
      for name in files:
         fullname = os.path.join(path,name)
         if os.path.exists(fullname):
            data = {
               'path' : path,
               'filename' : name,
               'size' : os.path.getsize(fullname),
               'ext' : os.path.splitext(name)[1],
               'mtime' : os.path.getmtime(fullname)
               }
            filedata.append(data)
   return filedata
```

Now, let's hook it up to Pandas and use it to analyze the Python source tree:

```
>>> import pandas
>>> filedata = pandas.DataFrame(summarize_files("Python-3.3.0rc1"))
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4207 entries, 0 to 4206
Data columns:
ext         4207  non-null values
```

```
filename    4207  non-null values
mtime       4207  non-null values
path        4207  non-null values
size        4207  non-null values
dtypes: float64(1), int64(1), object(3)
```

Let's see how many files of different types there are:

```
>>> filedata['ext'].value_counts()[:5]
.py   1618
.c    479
.rst  429
.h    263
.o    236
>>>
```

As a final example, let's generate a few statistics and use maplotlib to make a histogram. Here, we'll look at the sizes of .py files:

```
>>> pyfiles = filedata[filedata['ext'] == '.py']
>>> pyfiles['size'].max()
385802
>>> pyfiles['size'].mean()
12964.483930778739
>>> pyfiles['size'].std()
23799.089183395961
>>> pyfiles['size'].hist(bins=30)
<matplotlib.axes.AxesSubplot object at 0x102f0e290>
>>> import pylab
>>> pylab.show()
```

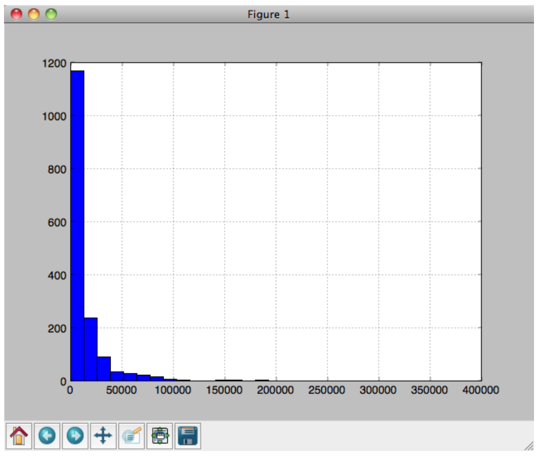


Figure 1: Histogram created with pyfiles['size'].hist(bins=30) using the matplotlib library

If it works, you'll end up with a plot that looks like Figure 1.

That's pretty neat—and it didn't involve much code.

Final Words and In Memoriam

If you're faced with the task of analyzing data, Pandas is definitely worth a look. Although all of the problems shown in this example could have been solved by short Python scripts, Pandas makes it even easier and more succinct.

Finally, in the last example, matplotlib (http://matplotlib.sourceforge.net) was used to make a plot. matplotlib is one of the most popular extensions to Python that is in widespread use by scientists and engineers. Sadly, John Hunter, the creator of matplotlib, passed away suddenly this past August from complications of cancer treatment, leaving behind his wife and three daughters. If you've benefited from the use of matplotlib, a memorial fund has been established. More information can be found at http://numfocus.org/johnhunter/.

iVoyeur
Nagios XI

DAVE JOSEPHSEN

Dave Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and is Senior Systems Engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project. dave-usenix@skeptech.org

I once had an excellent conversation with Puppet [1] creator Luke Kanies. Well, I think it was excellent but I don't remember exactly. It was LISA '07, I believe, and although I recall talking at length, all that remains in my memory is a vague notion that it was an excellent conversation, along with exactly two details. The first thing I remember is that I embarrassed myself in a particularly epic way. The topic was configuration management engines, and I think I made a quip about not knowing what I'd rather do less, write server configuration in XML or learn Ruby.

Luke, who had just wandered up, told me I should totally learn Ruby, because it was awesome. I responded that I'd recently read a series of articles about Ruby in *;login:* magazine [2], and although the author seemed to know what he was talking about, he hadn't managed to inspire in me a desire to run out and learn yet another OOP scripting language.

Luke, who had written those articles, apologized for not inspiring me.

The second thing I remember is his jacket, which did manage to inspire me. It was a really great jacket, and he wore it as if everyone just went around wearing awesome jackets all the time. As if he came from a land where awesome jackets were the most natural thing in the world and, having only recently arrived, hadn't yet caught on that we were—by our very nature—bereft of great jackets. The rest of us, he probably assumed, must have forgotten our awesome jackets at home or something. Seeing it made me want to be the type of person who could wear a jacket like that. More than that, I wanted to be the type of person who was capable of picking out for himself an awesome jacket to wear places, but, alas, I am not that type of person. Fundamentally, I think, it's a form vs. function thing.

Some of us have a talent for function. We buy ThinkPads, code in C and shell, and build low-level tools, or form higher level tools by combining simpler tools. Others excel at form: they buy MacBooks, code in Ruby, and make polished, shiny tools (with lots of library dependencies). A few of us even straddle the border, dabbling here and there and bringing form and function together.

It's a rare admin who, like me, tends toward function while coveting form. I am often confronted by the truth of this when I, knowing full well that a ThinkPad is what I need, talk myself into buying a MacBook. Like the jacket, it just doesn't fit. Meanwhile, I observe that my function-leaning friends are rarely afflicted with my fickleness. My friend Jeremy would run Linux on a ThinkPad if the keys were made of sandpaper and the pointer-nub was an upside-down thumbtack. My form-leaning friends, for their part, are even less likely to trade their iPhones for something with a (gasp) real keyboard.

Having been far less hung-over than the others in my party, I was in attendance at LISA '11 when Nagios creator Ethan Galstad was awarded the outstanding achievement award. Before he handed Ethan the award, David Blank-Edelman asked for the Nagios users in the room to raise their hands, and the response was easily 90th percentile. I was a little surprised by this because my impression had been that Nagios wasn't faring well with many sysadmins in the "form" crowd.

Don't get me wrong, the sysadmins who knew and loved Nagios were happy to see it continue in the way it always had, but its popularity had risen to the point that a different and more populous group of potential end-users had taken notice, and with them, Nagios doesn't seem to be comparing favorably with newer, prettier, and less flexible commercial competitors. Nagios was always a functional tool—a framework—and for those of us who lean toward function, it's usually enough that it's possible to make something prettier, that the pieces are all lying around.

This new breed of user seems to disagree. They have a few very specific gripes and are pretty vocal about them [3, 4] (and etc.). First, they find Nagios' configuration syntax unwieldy, to say nothing of the intolerable notion of (gasp) editing text files by hand. Second, they find the Nagios Web interface, with its C-based CGI and lack of pretty graphs, unforgivably old-fashioned. Finally, they seem to have no idea what to make of the fact that there is no database back-end. Jiminy Christmas, wrist watches and garbage disposals run MySQL these days! How is one to take seriously a monitoring system that doesn't?

For this considerable subset of users, Nagios' price tag ($0) doesn't make up for its abhorrent lack of bling, and answers to the effect that all of these things can be rectified with add-ons fall on deaf Bluetooth ear-pieces. Add-ons and hacks are birds in the bush, and they would rather pay for a bird in the hand than go beating around the bush themselves for free.

In the past few years, the Nagios people have therefore had an interesting problem to solve if they wanted to remain relevant: how to create a polished product that compares favorably with the likes of Zabbix, Hyperic, Patrol, Openview, etc. without destroying the extreme flexibility that makes Nagios what it is.

Enter Nagios XI

Nagios XI might best be called the perfect compromise between maintaining the power and flexibility of Nagios, while providing a turn-key monitoring system that more than satiates the desires of the PHP proletariat. But that description sells it short; XI is much more than just a shiny interface—it represents an enormous quantity of custom development and integration work. Further, there is real functionality in XI that simply can't be found in Nagios Core.

But neither can it be called a new monitoring system in its own right, because, in very important ways, it remains Nagios and retains all the flexibility and power on which so many of us have come to rely.

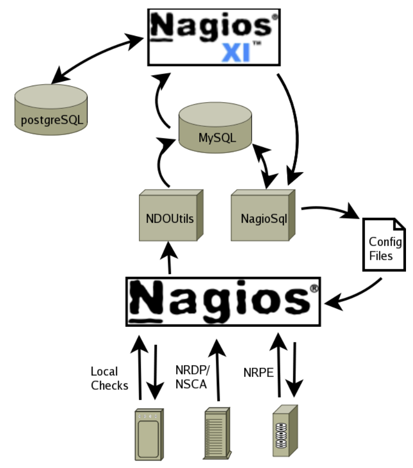


Figure 1: The Nagios XI architecture (simplified)

How Does It Work?

Figure 1 is a rough sketch of the Nagios XI architecture. As you can see, all of the actual host and service monitoring, as well as notification, escalation, etc., relies on an unmodified Nagios Core daemon, so any preexisting plugins or customization you might have will "just work" under XI. The NDOUtils plugin has been enabled and configured to replicate state information from Nagios Core into a MySQL database. Here is the primary information handoff between Core and XI; Nagios XI reads this database to glean information about the current state of hosts and services, as well as the Core daemon itself. This adds an information layer to Core that can be consumed by third-party UIs as well as your own custom integration scripts.

I've never been a fan of NDOUtils, but this is pretty much exactly the purpose for which it was created: to enable the development of custom Web UIs by giving Web developers (a notoriously database-focused lot) a database from which to glean state data.

NagiosQL, a popular add-on designed to add Web-based configuration to Nagios, provides the hooks necessary to modify the Nagios Core configuration from the XI interface. Every parameter that can be configured in the flat files may instead be set via the Web interface using the customized NagiosQL forms in the "Advanced Configuration" section of the XI interface. Although these forms are well integrated into XI, and retain an XI look and feel, there is a bit of a line in the sand between NagiosQL-driven core configuration, which is referred to as "advanced" in the XI interface, and the configuration parameters that are specific to XI itself.

This is because XI goes beyond presenting a simple Web wrapper to the Nagios Core configuration files, providing in addition a litany of semi-automated wizards and auto-discovery tools to ease the burden of initial and ongoing host and service configuration. I'll talk more about these in subsequent articles, but suffice to say that it is the intention of the XI creators to isolate the majority of XI users from the intricacies of the Nagios Core configuration to the extent that they never need to know what a check command is, much less a template. This is exactly what the form-crowd has asked for, and further, it makes it possible for monitoring configuration, traditionally an operations task, to be delegated to first-level support types, or in some environments, even to normal users. More clueful administrators who need to customize this or that can still do so, without editing the config files by hand, by using the NagiosQL-driven advanced configuration tool.

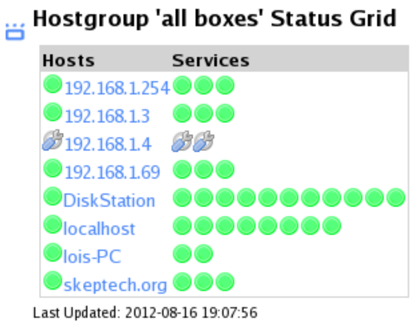
Configuration created by NagiosQL is automatically written to text configuration files in etc/nagios, and is read by the Core daemon from these flat files in the usual fashion. Although it's technically possible to hand edit these configuration files, it will avail you nothing because NagiosQL will eventually overwrite any changes you make. If you have your own configuration-generating automation (like Check_MK), or preexisting configuration that you do not wish to import into NagiosQL, or even if you're a curmudgeon who just prefers to edit the configuration manually, you can still maintain static config files in etc/nagios/static, and your files will still be parsed by the Core daemon while being left alone by NagiosQL. That runs both ways: statically configured hosts and services can't be modified via the UI unless you manually import them into NagiosQL (at which point they cease to be static).

Finally, Nagios XI maintains its own PostgreSQL database to store various configuration parameters, such as user-settings, custom dashboards, authentication info, and the like. Given not one but two database back-ends, the shiny new PHP interface, and the simplified configuration options, Nagios XI should satisfy the complaints I'm used to hearing from corporate administrators who are in the market for a "grown up" commercial monitoring product, but again, there's a lot more functionality than what I've encompassed in the architecture diagram.

What's In It for You?

One Slick Interface for starters. Given the general quality of the alternative PHP interfaces I'm used to finding in the Nagios Exchange repository, the XI interface is shockingly excellent. It is not yet another effort to bring the CGI interface "up to date" by replacing it with a PHP version of itself, but a complete rethinking of how the UI should work. It manages to take advantage of the strengths of a Web programming platform like PHP in a way that, like the jacket, awakens my repressed covetousness for style but, unlike the jacket, might actually fit.

Elements within dashboards can be moved around or deleted to suit the preferences of the user. AJAX is employed, both to update individual information elements and to provide feedback, so that when I send a command via the UI to reschedule a service check, or acknowledge an alert, a box momentarily appears to let me know my command has been accepted. One of my least favorite things about the Core UI is the way it dumps me to an acknowledgment page after I've issued a command, forcing me manually to navigate back to somewhere useful.



Hostgroup 'all boxes' Status Grid

Hosts	Services
● 192.168.1.254	●●●
● 192.168.1.3	●●●
◌ 192.168.1.4	◌◌
● 192.168.1.69	●●●
● DiskStation	●●●●●●●●●●●
● localhost	●●●●●●●●
● lois-PC	●●
● skeptech.org	●●●

Last Updated: 2012-08-16 19:07:56

Figure 2: Nagios XI minemap

Traditional Nagios tables like "service detail" and "hostgroup grid" still exist, but are implemented as repurposable widgets, which I can use to build custom dashboards. New tables have been added, a few of which are very dense and handy, such as the "minemap" visualization pictured in Figure 2.

Also included is integrated time series data (pretty graphs), a modularized component architecture with plugins such as the "Mass Acknowledgment Component" (yes acknowledge problems and schedule down time for groups of hosts and services), pretty new reports and interactive JavaScript-based visualization, a means of describing displaying and alerting on business process logic, auto-configuration and configuration wizards, and much more. All of which I will display for you like an awesome jacket in the next few articles.

Take it easy.

References

[1] The Puppet configuration engine: http://docs.puppetlabs.com/.

[2] Luke Kanies, "Why You Should Use Ruby," ;login:, vol. 31, no. 2, April 2006: https://www.usenix.org/publications/login/april-2006-volume-31-number-2/why-you-should-use-ruby.

[3] Nagios grumbling: http://www.frontlineops.net/2011/09/why-im-moving-from-nagios-to-zabbix.html.

[4] More Nagios grumbling: http://blog.desudesudesu.org/?p=1585.

/dev/random

ROBERT G. FERRELL

Robert G. Ferrell is a fourth-generation Texan, literary techno-geek, and finalist for the 2011 Robert Benchley Society Humor Writing Award. rgferrell@gmail.com

I had a dream about the Great Wall of China, as seen from space, which transmogrified after a few roof-reverberating snores into the Great Firewall of China, as seen from cyberspace. I woke up with my wife's pillow over my head and the urge to feature General Tso's chicken on my computer wallpaper—along with the germ of an idea. Since they haven't yet developed an effective antibiotic for the imagination (one might point to Hollywood's recent offerings in opposition to that assertion, but we'll carry on as though one had not), that germ quickly formed a slime-coated colony that took over an entire section of my brain the way the [insert your least-favorite major political party here] periodically take over Congress.

Moving cautiously but with all reasonable haste away from the politico-microbiological metaphor, I pondered over my first gallon of morning coffee what other Internet edifices could be said to be visible from cyberspace. First, though, I had to imagine something akin to the International (Cyber)Space Station drifting around on the Internet with little cybernauts peering out through the windows. They have tiny little cameras and even tinier little tubes of yogurt and I think I need another cup of coffee . . .

So, what else might said cybernauts spy as they endlessly circle the tube-within-a-tube (or whatever it was that silly politician said) that is the Internet? I came up with a few in the shower the next morning, although the scenario further degenerated from topological features to movies while I was shampooing my hair.

The Provider Backbone Bridge on the River Kwai: a group of captured network engineers is forced to build this structure that transports SONET-like connectivity along with carrier-grade packet-oriented traffic over the yawning chasm of Layer 2.

Router 66: Two geeks travel across America with a souped-up network analyzer looking for new protocols in the wild.

HUB: a network appliance breaks all the rules and goes renegade, assigning broadcast and collision domains at will just for kicks and hoarding all the bandwidth for itself. Eventually, the attached devices drop off, one by one, until only Hub is left with his little black book of MAC addresses, alone but still defiant.

I suppose that got a little silly; let's move on to more uplifting topics. My vote (the only one that counts, in this case) is for chatting about the place of Secure Engineering in the SDLC. First and most importantly, there is one. A place, I mean. Too many software developers seem to miss that simple notion. Not only is there a place, that place is most definitely not, as many more seem to think, in the "service pack two" stage. I feel an analogy coming on. I think it may even be a Grand Mal.

A popular, hip architect is hired to build a retail space. He decides, for ease of access and maximizing display effectiveness, to forgo any actual walls except for load-bearing beams. His design is met with overwhelming acclaim by the avant-garde architectural community. On opening day, customers flock to the store in droves, intrigued and exhilarated by the totally unfettered shopping experience. At first the concept seems brilliant, but then merchandise begins to disappear.

It's only a trickle at first . . . an item here and there. Not really noticed until the weekly inventory. Then the thefts escalate dramatically, until one morning the shelves are stripped bare. Nothing is left. Not a scrap. The architect is disheartened, as is, understandably, the store owner. They realize there must be something in place to prevent thievery, or at least make the thieves easier to detect and intercept. The architect hits upon the idea of trained geese.

The geese live two per cage, one cage per aisle, and sound the alarm if anyone comes into the store after closing hours. This works well for three nights. On the fourth morning the geese are gone and the butcher shop across the street coincidentally has a special on "pâté de foie gras, locally raised."

Undaunted, the architect next tries placing hidden tags on all the items in the store so that stolen merchandise can be more easily traced. This tactic is moderately successful until legitimate customers discover the tags and their function, at which point the goose poop hits the fan. Customers begin ripping their tags off and gluing them to the owner's car, house, clothing, and wife in protest. The store gets listed in a Web site for most egregious privacy offenders. Things are not going well for them, or for the architect.

In desperation, he hires armed guards with his own money to try to salvage the store. Sadly, the sight of them drives away customers in droves, until the business is no longer viable. The owner files for bankruptcy and the architect's career is badly stained by the whole sordid affair. He loses his license and is forced to move back in with his parents.

Some years later the now decaying premises, overrun with roaches, pigeons, and drug users, are bought by a young entrepreneur who immediately recognizes the enormous potential of the location. She painstakingly cleans and rebuilds, inside sturdy reinforced concrete walls with robust yet inconspicuous security measures included in the design from day one. She chooses well-constructed merchandise made with good quality materials, and offers it at reasonable prices. Her business is fabulously successful within mere weeks of opening, despite the walls. Customers do not seem to object to them at all, in fact, as they actually contribute additional space for displaying merchandise and allow the store to maintain a constant temperature as well as providing a safe, secure shopping environment. She eventually sells the store to a huge retail conglomerate and retires a multi-millionaire on her 42nd birthday.

Moral: Security belongs in the blueprints, not the remodeling plans.

Epilogue: I am proud to announce that after 40 years of avoiding it with one excuse or another, I have at long last taken the final step to full and unquestionable geekhood: HAM radio. You may now call me KF5SAR. General Class at present; hopefully by the time you read this, Amateur Extra.

QST, y'all.

NOTES

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration articles, tips and techniques, practical columns on such topics as security, Perl, networks, and operating systems, book reviews, and reports of sessions at USENIX conferences.

Access to *;login:* online from October 1997 to this month: www.usenix.org/publications/login/

Access to videos from USENIX events in the first six months after the event: www.usenix.org/conferences/multimedia/

Discounts on registration fees for all USENIX conferences.

Special discounts on a variety of products, books, software, and periodicals: www.usenix.org/member-services/discounts

The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.

For more information regarding membership or benefits, please see www.usenix.org/membership-services or contact office@usenix.org.
Phone: 510-528-8649

USENIX Board of Directors

Communicate directly with the USENIX Board of Directors by writing to board@usenix.org.

USA Team Wins Big at 2012 International Olympiad in Informatics

Brian C. Dean, Director, USA Computing Olympiad

Sirmione, a town on the shores of Lake Garda in Northern Italy, is normally a popular tourist destination for those who love beautiful scenery and excellent pasta. For one week in September 2012, however, this picturesque town was home to a special crowd: 310 of the best high-school computing students from around the world, who were representing their respective countries at the International Olympiad in Informatics.

Now in its 24th year, the IOI, one of several major international math and science Olympiads, is the world's most prestigious computing competition at the high-school level. Teams of up to four students arrive from each participating country (81 this year) to take part in a grueling two-day competition featuring some of the hardest algorithmic programming problems one can find. When the dust settles, top individuals are awarded gold, silver, and bronze medals according to their results. This year was one of the best ever for team USA, which won three golds and one bronze, just behind China and Russia with four golds each. Even better, USA student Johnny Ho achieved the only perfect score at the event, edging out superstar Gennady Korotkevitch from Belarus, winner of the previous three IOIs.

For team USA, the road to the IOI starts approximately one year before the event, with a series of monthly on-line programming competitions hosted by the USA Computing Olympiad. The USACO (usaco.org) is a non-profit organization funded by USENIX and other corporate sponsors, which provides training and online programming competitions for talented high school students at all levels, from novice students who have just learned to program to advanced students competing at the level of the IOI. Tens of thousands of students have participated in USACO training and competitions over the past decade alone. With high school computing education in the USA being in a somewhat lackluster state, the USACO plays a vital role in ensuring that bright students can continue to develop their skills beyond the standard high school computing curriculum.

Every year, the USACO invites the top 16 high school computing students nationwide to attend a rigorous academic summer camp, where they participate in advanced instruction in computational problem-solving techniques, excursions, enrichment activities, and, of course, plenty of ultimate frisbee. Every camp also features a game programming challenge for fun; this year, students teamed up to write programs to compete in a "blind man's bluff" poker game, which means you can see everyone's cards except yours. From USACO camp, the top four students are selected to represent the USA at the IOI. This year, team USA consisted of:

- Johnny Ho (gold medal, 1st place winner), from Lynbrook High School in California
- Mitchell Lee (gold medal), home-schooled in Virginia
- Scott Wu (gold medal), from Baton Rouge Magnet High School in Louisiana
- Daniel Ziegler (bronze medal), from Davidson Academy in Nevada

Traveling to the IOI in Italy with the team this year were team leader and USACO director Brian Dean and deputy leader Jacob Steinhardt. Brian is a computer science professor at Clemson University, and Jacob recently graduated from MIT and is beginning his PhD studies in computer science at Stanford under the direction of new Stanford professor Percy Liang, also a USACO alum and previous IOI medalist.

In addition to two full days of competition, IOI participants were treated to the best of Italian culture and cuisine, and excursions to Milan and Venice. IOI 2012 was an unforgettable experience. We look forward to another strong showing for team USA next summer at IOI 2013 in Brisbane, Australia.

Thanks to Our Volunteers

Anne Dickison and Casey Henderson, USENIX Co-Executive Directors

As many of our members know, USENIX's success is attributable to a large number of volunteers, who lend their expertise and support for our conferences, publications, good works, and member services. They work closely with our staff to bring you the best there is in the fields of systems research and system administration. Many of you have participated on program committees, steering committees, and subcommittees, as well as contributing to this magazine. We are most grateful to you all. We would like to make special mention of some people who made particularly significant contributions in 2012.

First, we would like to offer special thanks again to Margo Seltzer, President of the USENIX Board of Directors, for her service as Acting Executive Director of USENIX until we took over the reins in April of this year.

Program Chairs

William J. Bolosky and Jason Flinn: 10th USENIX Conference on File and Storage Technologies (FAST '12)

Steven Gribble and Dina Katabi: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)

Olivier Bonaventure and Ramana Kompella: 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE '12)

Engin Kirda: 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET '12)

Hans-J. Boehm and Luis Ceze: 4th USENIX Workshop on Hot Topics in Parallelism (HotPar '12)

Gernot Heiser and Wilson Hsieh: 2012 USENIX Annual Technical Conference (USENIX ATC '12)

Michael Maximilien: 3rd USENIX Conference on Web Application Development (WebApps '12)

Rodrigo Fonseca and Dave Maltz: 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '12)

Raju Rangaswami: 4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '12)

Christopher Walsh: 2012 USENIX Workshop on Hot Topics in Cyberlaw (USENIX Cyberlaw '12)

Daniel V. Klein: 2012 USENIX Configuration Management Summit (UCMS '12)

Nicole Forsgren Velasquez and Carolyn Rowland: 2012 USENIX Women in Advanced Computing Summit (WiAC '12)

Umut A. Acar and Todd J. Green: 4th USENIX Workshop on the Theory and Practice of Provenance (TaPP '12)

Kameswari Chebrolu and Brian Noble: 6th USENIX/ACM Workshop on Networked Systems for Developing Regions (NSDR '12)

Tadayoshi Kohno: 21st USENIX Security Symposium (USENIX Security '12)

J. Alex Halderman and Olivier Pereira: 2012 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE '12)

Sean Peisert and Stephen Schwab: 5th Workshop on Cyber Security Experimentation and Test (CSET '12)

Roger Dingledine and Joss Wright: 2nd USENIX Workshop on Free and Open Communications on the Internet (FOCI '12)

Carl Gunter and Zachary Peterson: 3rd USENIX Workshop on Health Security and Privacy (HealthSec '12)

Patrick Traynor: 7th USENIX Workshop on Hot Topics in Security (HotSec '12)

Anton Chuvakin: Seventh Workshop on Security Metrics (MetriCon 7.0)

Elie Bursztein and Thomas Dullien: 6th USENIX Workshop on Offensive Technologies (WOOT '12)

Chandu Thekkath and Amin Vahdat: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)

Michael J. Freedman and Neeraj Suri: Eighth Workshop on Hot Topics in System Dependability (HotDep '12)

Mani Srivastava and Thomas F. Wenisch: 2012 Workshop on Power-Aware Computing and Systems (HotPower '12)

Peter Bodik and Greg Bronevetsky: 2012 Workshop on Managing Systems Automatically and Dynamically (MAD '12)

Dilma Da Silva, Jeanna Matthews, and James Mickens: 2012 Workshop on Supporting Diversity in Systems Research (Diversity '12)

Carolyn Rowland: 26th Large Installation System Administration Conference (LISA '12)

Invited Talks/Special Track Chairs

James Mickens, Florentina Popovici, and Jiri Schindler: Posters and Work-in-Progress Reports (WiPs) Committee at FAST

John Strunk: Tutorial Chair at FAST

Srikanth Kandula: Poster Session Coordinator at NSDI

Emil Sit: Poster Session Coordinator at USENIX ATC

David Evans, David Molnar, Bruce Potter, and Margo Seltzer: Invited Talks Committee at USENIX Security

Matt Bishop: Poster Session Coordinator at USENIX Security

Matt Blaze: Rump Session Chair at USENIX Security

Shan Lu and Junfeng Yang: Poster Session Co-Chairs at OSDI

Narayan Desai, Cory Lueninghoener, and Kent Skaar: Invited Talks Coordinators at LISA

Lee Damon: Lightning Talks Coordinator at LISA

Kyrre Begnum: Workshops Coordinator at LISA

Chris St. Pierre: Guru Is In Coordinator at LISA

Marc Chiarini: Poster Session Coordinator at LISA

Other Major Contributors

John Arrasjid, David Blank-Edelman, Sasha Fedorova, Matt Blaze, Clem Cole, Alva Couch, Brian Noble, Niels Provos, Carolyn Rowland, Margo Seltzer, and Dan Wallach for their service on the USENIX Board of Directors

Dan Geer, Eric Allman, and Niels Provos for serving on the Audit Committee

Alva Couch for chairing the USENIX Board of Directors Nominating Committee

Brian Dean, Mark Gordon, Rob Kolstad, Richard Peng, Don Piele, Eric Price, Jacob Steinhardt, and Neal Wu, this year's directors and coaches for the USA Computing Olympiad, co-sponsored by USENIX

Eddie Kohler for his HotCRP submissions and reviewing system

Jacob Farmer of Cambridge Computer for his sponsorship of the traveling LISA Data Storage Day series and for organizing the Storage Pavilion and Data Storage Day at LISA '12

Mark Burgess and Diego Zamboni for writing the Short Topics book published by USENIX in 2012

Matt Simmons, Ben Cotton, and Greg Riedesel for blogging about USENIX and LISA '12 activities

Book Reviews

ELIZABETH ZWICKY, WITH RIK FARROW

Illustrated Guide to Home Forensic Science Experiments: All Lab, No Lecture
Robert Bruce Thompson and Barbara Fritchman Thompson
O'Reilly Media, July 2012. 428 pages.

ISBN 978-1-449-33451-2

This is a careful, practical guide to doing real forensic science at home. Kind of like those detective kits they try to sell kids, except with practical advice on really lifting fingerprints, including superglue fuming techniques. (In case you were wondering, the reason the detective kit didn't work is that dusting for fingerprints successfully is hard.) It's not at all like television; there's actual work involved.

On the one hand, it is just as cool as it sounds. On the other hand, did I mention it's not like television? I'd be prepared for the cool factor to wear off pretty quickly for the younger set. Doing science—forensic or not—requires a certain amount of meticulous attention to detail, accurate performance of repetitive processes, and a willingness to embrace "Yes, those are the same shade of orange" as a fabulous result.

But if you're willing to pay careful attention to detail, you too can do gel electrophoresis of DNA at home using primarily things available at your local supermarket (well, my local supermarket, which sells chopsticks and agar as well as Tupperware and D batteries). I have to admit I didn't do the experiments, but I did find myself with a sudden desire for a microscope so that I could.

The text is interesting and neither undersells nor oversells the experiments. It honestly lays out the authors' experiences with the practical issues. My only gripe is that some of the images are poor, and in many of them the authors' hands are shown ungloved when the text calls for gloves.

OS X Mountain Lion: The Missing Manual
David Pogue
O'Reilly Media, July 2012. 834 pages.

ISBN 978-1449-33027-9

OS X Mountain Lion Pocket Guide
Chris Seibold
O'Reilly Media, July 2012. 236 pages.

ISBN 978-1-449-33032-3

I have just upgraded to Mountain Lion, making a dizzying leap from running Snow Leopard on the home machine and Leopard on the work machine to Mountain Lion and Lion, so it seemed like a good moment to check out some Mountain Lion books. Despite the hoopla about Lion, as a very long-time Mac and iPad user, I didn't actually find anything particularly earth-shattering about it, or about Mountain Lion, either.

Then I picked up *Mountain Lion: The Missing Manual* and discovered that this was because I just didn't know about the exciting new bits. I'm still not convinced that they will change my life, but I am intrigued at the possibility that gestures will finally make full screen mode and spaces work for me. And despite my functional knowledge of Macintoshes, I learned useful things (Spotlight will do calculations for you, which gives you a keystroke that brings up a calculator that doesn't obscure the numbers you wanted to add—sadly, it is then impossible to cut-and-paste the answer). *The Missing Manual* also is smoothly readable and mildly funny (which, trust me, is all the funny you are likely to put up with for 834 pages). I'm not sure that I would be motivated to read it if I weren't reviewing it, but that's not its fault; I just don't have any driving desire to know that much all at once about my operating system. And it's obsessed with keyboard shortcuts, which I have a very limited amount of use for. If you want to know all the details instead of guessing, and/or you hate using a mouse, it's an excellent choice.

The *Mountain Lion Pocket Guide* is more my speed; the keyboard shortcuts are in sidebars and their own chapter instead of spread throughout, and it's brief. On the other hand, it's also slower to get to the cool stuff, or at least the stuff I think is cool. Its 10 cool new Mountain Lion features, although they are new to Mountain Lion, fail to pass my coolness bar reliably, and oddly it disagrees with *The Missing Manual* about one new feature; *The Missing Manual* says Mountain Lion now does full-screen separately on multiple screens, *Pocket Guide* says it does not. Sadly, *Pocket Guide* is right. Either one will let me have one app full screen on either monitor, while the other monitor displays a picture of linen apparently designed to convey blankness while letting you know the power is still on.

If you aren't particularly feeling the lack of a manual, you probably won't love either book. If you are feeling the lack of a manual in a nagging way, you probably want the *Pocket Guide*. If you'd like to cuddle up with a manual and have the interfaces to everything lovingly but gently explained to you, go for *The Missing Manual*. (This is also a good choice if you would like to feel clever and superior at work, where people are actually quite impressed by what I picked up.)

Think Like a Programmer: An Introduction to Creative Problem Solving

V. Anton Sproul

No Starch Press, 2012. 227 pages.

ISBN 978-1-59327-424-5

This is a valiant effort at teaching somebody who is intelligent and motivated how you do the essential parts of programming. Not what "if . . . then . . . else" means, or how you write syntactically correct code in some language, but how to program. As a reviewer, I'm handicapped by the fact that I've always known how to do that. I have, however, spent time teaching people to solve problems, with some success.

I think on the whole this is a worthy approach; it tries to demystify the process and break it down into pieces, which is very useful for learners. In fact, merely existing is a good first step, because many people—including learners—have a firm belief that these things can't be taught, which doesn't help.

I have two quarrels with the book. First, I think it moves over a lot of territory too fast. Yes, there are exercises, and people are strongly encouraged to do them. But exercises by themselves are not enough for most people learning to solve problems; they need smaller bites at a time. Additionally, some of these exercises don't appear to me to be possible as stated. For instance, using only two output statements, one that outputs the hash mark and one that outputs an end-of-line, writes a program that produces the following shape:

```
########
 ######
  ####
   ##
```

Where did those initial spaces at the beginning of the line come from? We don't get to output spaces. Is there some C++ trick here I don't know about? If so, why are we depending on it in the first chapter where we actually write programs? This just seems mean, one way or another.

Second, it's not clear to me why the author thinks C++ is a good choice. Sure, that allows you to spend a lot of time thinking about linked lists and low-level programming abstractions, but it doesn't allow you to spend a lot of time thinking about appropriate data structures for your problem and high-level programming abstractions other than classes. Linked lists are not actually a good answer to most programs these days. Use a database. Don't need a database? Use a hash. I am not convinced that knowing how to do linked lists and think about pointers will teach a junior programmer how to avoid linked lists when they are not needed, which is just about always.

Essential Scrum: A Practical Guide to the Most Popular Agile Process

Kenneth S. Rubin

Pearson Education, July 2012. 426 pages.

ISBN 978-0-13-704329-3

This lives up to its title; it lays out, clearly, the basics of Scrum, from reasoning through mechanics. It's a fairly minimal version of Scrum; it doesn't insist on pair programming or user stories, for instance, and it doesn't try to deal with edge cases. There's a paragraph or two on Scrum in interrupt-driven groups (summary: don't), but nothing about other common issues (Scrum surrounded by other theories, groups divided by distance). If you are coming into a solidly Scrum-based organization, this would be a good way to get a feel for the bones of Scrum, separate from organizational issues and peculiarities. If you're struggling with a transition to Scrum, *The Scrum Field Guide* (previously reviewed) is a better choice. It's more dogmatic about some particular Scrum features, but much stronger on the nitty-gritty of transitioning to Scrum and running Scrum in edge cases. If you have patience for two books, they make a good pair and illuminate some of the internal variation in what Scrum practitioners find important.

—Elizabeth Zwicky

LED Lighting: A Primer for Lighting the Future

Sal Cangeloso

O'Reilly Maker Press, 2012. 58 pages.

ISBN 978-1-449-33476-5

As the time has come to phase out the terribly simple and horribly inefficient 100-watt incandescent bulbs, I wanted to learn more about LED-based lighting. I've already done most of the home improvements possible for a home: extra insulation in the attic, reflective barriers, extra insulation added to outside walls, 3.2 kW of solar panels, double-pane windows, and no incandescent bulbs in my house. Since I've already embraced fluorescents, what do I need to know about LEDs?

Cangeloso informs us of issues involving four forms of lighting: incandescents, halogen, fluorescents, and LEDs. I had already bought a couple of LED bulbs as experiments, and they were pretty disappointing: dim and off-color. And what's with the enormous heatsinks found in floodlight-type LEDs?

Lighting is heavily regulated, both to prevent fires and to provide standards for consumers. Whereas we once bought incandescent bulbs based on wattage, this is a poor method when it comes to more efficient bulbs measured in lumens. I still find myself comparing wattage to lumens. And it turns out that the old incandescent bulbs have another endearing feature: they produce a fuller spectrum of warm lighting. This isn't much of a surprise when you have a filament that glows white hot inside a protective enclosure: you get everything from ultraviolet to lots of infrared light. Cangeloso explains color temperatures that helped me understand the "cool" and "warm" terms.

LEDs, and incandescents, have another problem, and that is color accuracy when used for lighting. The warm, full spectrum is closer to what our eyes have evolved to use, but LED bulbs, which actually do use phosphors to produce most of their light, have difficulty producing anything close to a full spectrum. The color rendering index (CRI) for an incandescent bulb is 100, 80 for Philips' most popular A19 (the familiar screw-in base) bulb, and 92 for its award-winning (and very expensive) bulb. Color temperature is going to be more important for most people, but not for artists and other people who expect or require their colors to appear true.

What this book lacks are tables. I found myself paging back through the book for the watts-to-lumens comparisons, and just wished there was at least one table in this short book.

And those heatsinks? It turns out the LEDs produce less light (and don't last as long) when they get hot, and while incandescents discard lots of heat as infrared light, LEDs need other forms of cooling. That, and because LED bulbs also have electronics in their bases that provide both regulated power and reduced power if overheating is detected, makes LED bulbs truly high-tech compared to the glowing filaments that are now being hoarded by some. I can suggest reading this book if you want to learn more, see cutaway photos of LED bulbs, and want to prepare for the future.

—Rik Farrow

REPORTS

Conference Reports

In this issue:

Conference reports from USENIX Security '12, WOOT '12, HotSec '12, CSET '12, HealthSec '12, and EVT/WOTE '12 are online at: www.usenix.org/publications/login

2012 Electronic Voting Technology Workshop/ Workshop on Trustworthy Elections (EVT/ WOTE '12)

Bellevue, WA
August 6–7, 2012

New Interfaces

Summarized by Harvie Branscomb (harvie@electionquality.com)

Operator-Assisted Tabulation of Optical Scan Ballots

Kai Wang, University of California, San Diego; Nicholas Carlini, Eric Kim, Ivan Motyashov, Daniel Nguyen, and David Wagner, University of California, Berkeley

Kai Wang discussed OpenCount (code.google.com/p/opencount), an open-source software project to productively combine human cognition with machine functionality for the purpose of tabulating scans of paper ballots. The design was motivated by a need to go beyond what software can do to interpret interesting cases, such as poor markings and erasure marks. OpenCount interleaves computer vision techniques with focused operator verification to produce a "cast vote record" of each scan of a ballot suitable for performing "single-ballot level" risk-limiting comparison audits. The system does not rely on ballot vendor specifications or definition files and may be used with existing scans of voted ballots, but it requires a not-voted or "blank" instance of every unique ballot style used in the election for configuration.

The first phase involves an interaction with an operator to identify a rectangular area around a voting "target" that the system uses to find similar others, automatically grouping them into clusters. Portions of text or image are operator selected to identify and classify each unique style, such as party, language, precinct, etc. In the second phase, voted ballot scans are spatially translated and rotated as necessary for registration with the data from the unvoted examples at the pixel level across the entire ballot, and then again for each voting target. Each target in every voted ballot is displayed in an array ordered by average pixel density to allow the operator to inspect visually and determine the threshold between marks to be classified as votes and those not to be so classified.

OpenCount has been successfully validated in California counties through secondary scanning of several manufacturers' styles of ballots in five risk-limiting audits in 2011 and two in 2012, with at least four more upcoming. Election officials have agreed that OpenCount provides a more accurate count than purely machine counts not using operator input.

Anna Queredo asked how the system notifies the operator which ballots to look at. Kai explained that the operator scrolls to the border between marked and unmarked targets and focuses attention there. He said there is also a function to handle ballot scans for which something unusual happened separately. Jeremy Epstein asked how to notice marks that are outside the "target." Kai explained that marks within a defined rectangular area surrounding the "target" are recognized, but everything outside of that rectangle is ignored.

A Hybrid Touch Interface for Prêt à Voter

Chris Culnane, University of Surrey, Trustworthy Voting Systems Project

Prêt à Voter (pret ah votay) in its original form is an end-to-end verifiable paper-ballot voting system design that is machine tabulated such that no machine learns what the voter intent is; thus, it systematically retains the privacy of the vote. The design's central element is a paper ballot that can be split so that, after the ballot is marked, the randomly ordered list of candidates becomes separated from the voter's marks. A crypto key containing a signed serial number protects access to the knowledge of the order of the candidates on the ballot while the marks themselves remain public. Chris Culnane's talk introduced an accessibility extension of Prêt à Voter in which the right-hand side of the ballot, the portion to be marked, is implemented on a touch screen such that all of the integrity features of the original design are maintained while additional accessibility features such as tactile and auditory cues could be implemented.

Two implementations were described, one for the original Microsoft Surface interactive desktop and another for a 3M Multi-Touch M2256PW. Chris also speculated about a third using the Samsung SUR40 with Microsoft PixelSense. In the design, the surface of the screen must recognize a 2D barcode or a coded conductive ink or foil that (1) informs the system of the location and orientation of one or more paper left-hand ballot sides containing lists of human-readable ballot choices (e.g., candidates) and (2) allows the screen to display the right-hand side(s) of the ballot as indistinguishable vote targets in the appropriate location(s). In Chris' implementation only the left-hand side of the Prêt à Voter ballot exists in paper form.

Chris admitted to concerns that voters might believe the system could recognize their face through the glass, although the technology does not have that capability. Philip Stark asked about the time-frame to deploy. Chris admitted that he has limited access to the necessary equipment and much work is yet to be done. Jeremy Epstein asked about the range of disabilities that could be served by this system. Chris said the system only requires the ability to place the left-hand side of a Prêt à Voter paper ballot on the display surface and read it; thus embossing such as Braille may be needed. Also, the system must hold the paper in place, so this limits the extent to which the display can be placed vertically. Flexibility in orientation is advantageous to voters with disabilities. Chris suggested that a move to a smaller form factor would help broaden the scope of application to various disabilities, and price is also a concern. Peter Neumann questioned the need for trust of the underlying technology, but Chris reassured him that the system does not learn the permutations of the candidate order; instead the system has access to only a serial number protected by cryptography.

Election Auditing

Summarized by Harvie Branscomb (harvie@electionquality.com)

A Bayesian Method for Auditing Elections

Ronald L. Rivest and Emily Shen, Massachusetts Institute of Technology

Ron Rivest brings new resources from the field of statistics to the practice of auditing elections. His talk about a "ballot-polling" method of auditing described the use of Bayesian methods and their multiple advantages, including for more familiar "comparison audits." Ballot polling does not require access to data from a voting system and instead independently predicts the likelihood that any given candidate would be declared the winner after counting all of the ballots while usually counting relatively few. It does require the ability to randomly select and interpret the voter intent on every ballot marked by every voter in an election contest. While the method is easy to describe on a single page, the extent of the calculations needed requires the assistance of a machine for most elections. The Bayes audit does not require knowledge of the margin of victory and conveniently permits multiple auditors with multiple "Bayes priors" to be accommodated. Bayes priors can reflect real biases among interested parties, such as the expectations of a losing candidate who believes that uncounted ballots are voted in his or her favor.

Ron reported that Bayes audits offer good efficiency, comparable to that found in Stark's ballot-polling and single-ballot comparison-audit methods. He said Bayes methods can also be applied to comparison audits, which offer even better efficiency over ballot polling. Many voting methods can be supported. Small and controllable miscertification rates are observed. Even if the audit is stopped early for practical reasons, meaningful results can be obtained. Disadvantages include applicability only to single-ballot audits with results depending somewhat on the choice of prior. How Bayes audits relate to risk-limiting audits remains an open question.

David Flater was concerned about non-obvious stopping criteria and the need to control the risk to a specified level. Ron Rivest explained that the analysis involved in the Bayes audit is nicer than with other methods where it gets complicated but that risk measures depend on the priors. Bayes methods represent a solid approach to, for example, financial audits. Peter Neumann expressed concern that this method has the potential to predict election outcomes from incomplete data and also that IRV outcomes could not be calculated without complete data. Ron Rivest explained that, for all voting methods, you need everything "in" before beginning the audit. John Bodin talked about comparison audits and the need for an identifier to connect ballots to interpretations and the security of this identifier. Philip Stark commented that part of an election is convincing the loser that they have a "prior."

More information is available at: people.csail.mit.edu/rivest/bayes.

BRAVO: Ballot-polling Risk-limiting Audits to Verify Outcomes

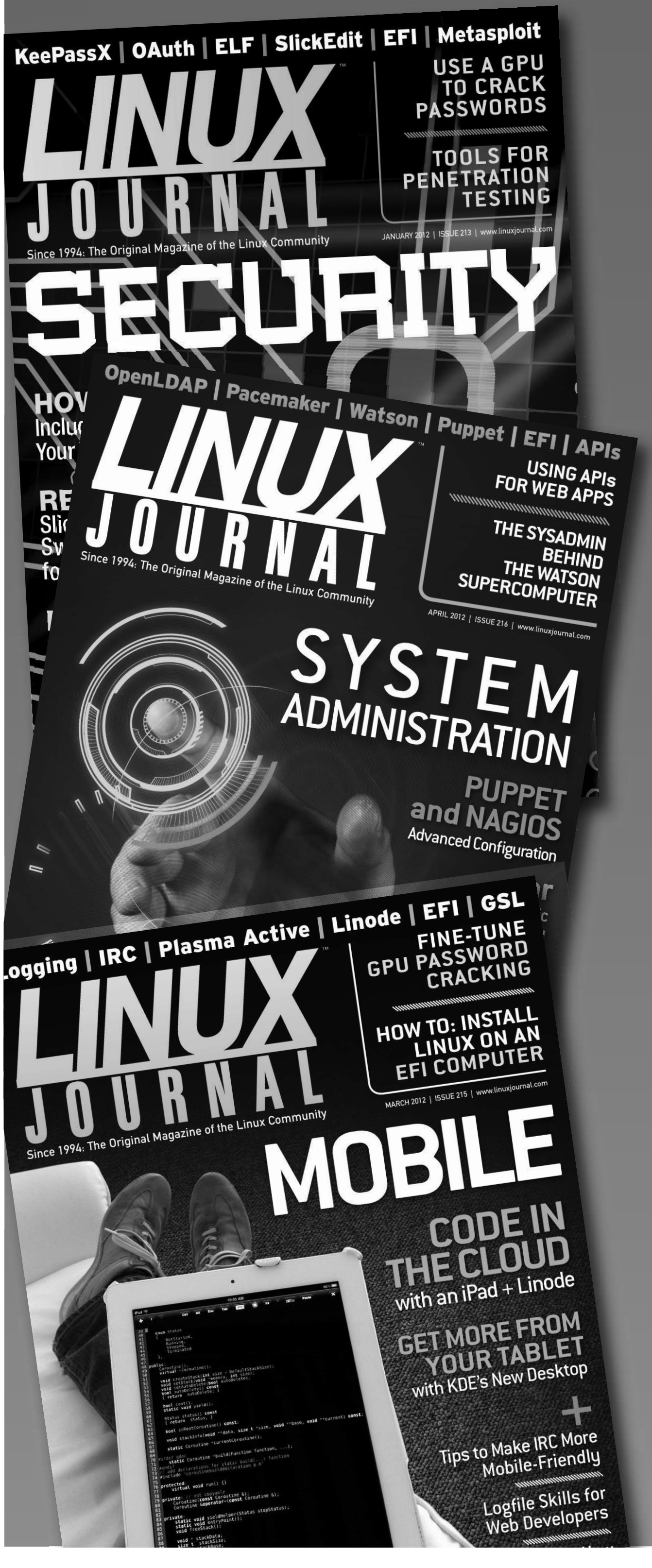
Mark Lindeman, Philip B. Stark, and Vincent S. Yates, University of California, Berkeley

Philip Stark (statistics.berkeley.edu/~stark/Vote) opened by emphasizing the importance of evidence-based elections and the critical need for an adequate evidence trail measured by a compliance audit as a prerequisite to any successful election audit. Stark has been conducting single-ballot risk-limiting audits both in comparison-audit format and the more recently proposed ballot-polling mode. In a risk-limiting audit, the question is not how many ballots to audit at first, but when to stop. If there is compelling evidence that the outcome is correct, then stop; otherwise continue the audit, ballot by ballot, tabulating the result incrementally at each step. In ballot-polling audits, you hand count votes, whereas in comparison audits, you count the discrepancies between a machine count and a hand count. According to Philip, in defining the "risk" of a risk-limiting audit, one assumes that a reported outcome might be wrong in the most maliciously difficult way to detect. The "risk" is the chance that this wrong-outcome scenario would not be detected and would not be corrected by the audit. This is quite different from the risk that any outcome is wrong. Numerous risk-limiting audits have now been conducted by Stark et al. in California elections ranging in size from 200 ballots to 121,000 ballots.

Ballot-polling audits require more ballots to be audited than equivalent comparison audits, but do not require any results from the voting system and have no setup costs such as the need for secondary scanning of ballots. Although polling audits do not check the voting system tabulation, they do expose the voter marks on only relatively few sampled ballots and a number comparable to that of a precinct comparison audit. A good ballot manifest is needed in order to be able to select a random sample, but the method can be executed with dice and a pencil and paper if desired.

The methodology is reminiscent of a public opinion poll where the ballot is asked, "What do you say?" Philip Stark reported several successful election audit experiences in California. He then extrapolated the audit workload for the average statewide presidential contest. Among 255 statewide presidential contests between 1992 and 2008, the median expected sample size for a statewide ballot-polling audit would be only 307 ballots.

Douglas Wikström suggested a potential for avoiding sequential sampling by using simultaneous multiple ballot sampling or even some special handling of the ballot in the voting booth such as "tossing a p coin." Philip agreed some potential benefit might result. Peter Neumann asked about exit polls. Philip Stark replied that exit polls are a biased sample, encounter problems with people's willingness to answer accurately, and are in effect a mess.


KeePassX | OAuth | ELF | SlickEdit | EFI | Metasploit
LINUX JOURNAL
USE A GPU TO CRACK PASSWORDS
TOOLS FOR PENETRATION TESTING
Since 1994: The Original Magazine of the Linux Community
JANUARY 2012 | ISSUE 213 | www.linuxjournal.com
SECURITY
OpenLDAP | Pacemaker | Watson | Puppet | EFI | APIs
USING APIs FOR WEB APPS
THE SYSADMIN BEHIND THE WATSON SUPERCOMPUTER
APRIL 2012 | ISSUE 216 | www.linuxjournal.com
SYSTEM ADMINISTRATION
PUPPET and NAGIOS Advanced Configuration
Logging | IRC | Plasma Active | Linode | EFI | GSL
FINE-TUNE GPU PASSWORD CRACKING
HOW TO: INSTALL LINUX ON AN EFI COMPUTER
MARCH 2012 | ISSUE 215 | www.linuxjournal.com
MOBILE
CODE IN THE CLOUD with an iPad + Linode
GET MORE FROM YOUR TABLET with KDE's New Desktop
Tips to Make IRC More Mobile-Friendly
Logfile Skills for Web Developers

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

SAVE THE DATE!

FEB. 12–15, 2013 • SAN JOSE, CA

FAST '13

11th USENIX Conference on File and Storage Technologies

FAST '13 brings together storage-system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems. The conference will consist of technical presentations, including refereed papers, Work-in-Progress (WiP) reports, poster sessions, and tutorials.

Full program information and registration will be available soon.
www.usenix.org/conference/fast13

Sponsored by USENIX in cooperation with ACM SIGOPS