JEFFREY BERG, EVAN TERAN, AND SAM STOVER

investigating Argos

Jeffrey Berg manages the Public Vulnerability Research Team at iSIGHT Partners. His primary interests are network security and penetration testing.

jbergcosc@gmail.com

Evan Teran is the Lead Security Researcher for iSIGHT Partners, a startup which produces human and electronic intelligence-fused products and services. His primary interests are in the fields of reverse engineering and operating systems.

evan.teran@gmail.com

Sam Stover is the Director of Tech Ops for iSIGHT Partners. His research interests include detection and mitigation methods for vulns and malware.

sam.stover@gmail.com

ARGOS IS A HIGH-INTERACTION HONEYPOT built for new vulnerability discovery—or, to be more accurate, "zero-day vulnerability in use" discovery. We cover the background and architecture of the program, the underlying concepts and the overall functionality, as well as the output. The strengths and weaknesses of Argos are examined, and we provide a brief overview of what is necessary to set it up in a realistic scenario to capture exploitation information and, possibly, new vulnerability information. Finally, we introduce an independently developed tool capable of parsing the output generated by Argos.

A honeypot is a network decoy. It can serve several purposes, including distracting malicious actors from valuable machines, providing near-real-time intelligence on attack and exploitation trends, and giving advance notice of potential new, "zero-day" vulnerabilities [1]. The security researcher configures a vulnerable machine that has the look and feel of a real service, a set of services, or the entire system on which he or she would like to gather exploitation information. A set of concealed processes can exist underneath this configuration which monitor and log everything from network activity to memory usage to program execution flow. The key to a good honeypot configuration is logging only the data related to a "compromising event" or the events immediately surrounding the compromise. The rest of the information comprises extraneous data that adds to the time required for the researcher to find and aggregate the information related to the actual compromise. A good source for a range of honeypot topics from background information to technical implementation guidance is Provos and Holz's *Virtual Honeypots: From Botnet Tracking to Intrusion Detection* [2].

Argos

Argos [6] can detect zero-day vulnerabilities when used as a tool by a vulnerability researcher. A hybrid setup of an updated Intrusion Detection System in front of an Argos system would likely be the best configuration for confirming such a situation; however, that falls outside the scope of this article.

Argos is based on and extends the functionality of QEMU, an open source processor emulator. QEMU

uses dynamic translation to provide users with the ability to run virtual machines of one architecture on a host system of a different architecture [3]. Argos was initially developed as a patch for QEMU to extend functionality (adding the ability to detect attempts to compromise the emulated guest). However, it has since morphed to incorporate QEMU into the overall structure. Essentially, if Argos were installed alone, it could run a virtual image independently. However, QEMU is needed for creating images in addition to easy and fast administration of those images. Also, because some unpackers (such as the one used by Windows XP SP2) cause false positives, it is easier to administer the honeypot using QEMU.

The main technique used by Argos to detect the point in time in which a malicious actor compromises a guest system is called "dynamic taint analysis." Dynamic taint analysis operates on the precept that all external input to a honeypot is malicious or tainted. Thus, Argos tracks all external input by marking it as well as any variables or registers the initial input is involved in modifying, either directly or indirectly. For example, consider the MOV operation:

```
MOV AX, DX
```

If DX is tainted, AX will be tainted after the operation executes. It follows that variables and registers touched by AX further down the execution path will then be tainted. If any of these tainted elements becomes involved in changing the flow of program execution, Argos logs the information in the form of a special memory dump.

Again, Argos was designed for detecting new zero-day vulnerabilities in the wild and, for the most part, it is successful in doing so. We programmed several types of vulnerabilities into test images or installed vulnerable programs. Argos detected exploitation on almost all of them, including stack-based buffer overflow vulnerabilities, heap-based buffer overflow vulnerabilities, and format string issues. A specific exploitation method that is not recognizable by Argos will be discussed later in this article.

When Argos detects an attack, two events occur on the host machine: Argos will perform the memory dump into a .csi file and two or more lines are written to standard output that look like this:

```
[ARGOS] Attack detected, code <CI> PC <10111ac> TARGET <12ffb0>
[ARGOS] Log generated <argos.csi.1719845868>
[ARGOS] Injecting forensics shellcode at 0x00131000[0x08e28000]
```

The first tells the researcher that an attack has been detected, along with a two- or three-letter description such as JMP. The second line tells the researcher the file in which the memory dump may be found. The .csi file is placed in the directory from which Argos was launched. We created a couple of scripts to configure the network as well as to launch Argos from date-stamped directories, so the .csi files were placed in these directories. We found this very useful for organizational purposes. Optionally, Argos can also inject forensic shellcode into the process to try to extract extra useful information, as indicated by the third line.

The .csi file is created when the flow of execution is altered as a result of a tainted variable. In the .csi file, Argos will save the memory associated with the targeted process as well as the variables tainted by external input. An especially vital portion of this dump, which was added in version 2 of the .csi files, is the last good instruction pointer address. A researcher could use this address later to reverse the vulnerable process and find the exact vulnerable function and root vulnerability.

Installing and Setting Up Argos

Installing Argos is relatively straightforward. Various instruction sets exist that differ on the level of detail provided, most likely because each set of instructions considers different versions of QEMU, SDL, and, most important, Argos. As of this writing, the current versions of QEMU and Argos were 0.9.1 and 0.4.1, respectively. The latest source may be downloaded via the Vrije Universiteit site [6]. In addition to QEMU and Argos, the Simple Direct Media Layer Library is needed, while KQEMU (the accelerator module, not the front end) is optional. Since Argos does not use this accelerator, the only benefit of KQEMU is the decreased time necessary to create and

manage images with QEMU. With respect to hardware requirements, we recommend at least 1 GB of RAM, a sizable hard drive dependent upon the number of images you would like to create and store, and two Network Interface Cards (NICs). We suggest two NICs so that one interface can be used as a management interface while the second interface can be used to expose the honeypot to malicious or potentially malicious traffic. In a later section we will touch on deployment options and their impact on the type of traffic received. Furthermore, for security purposes it may be wise to configure the management interface on a separate network segment, away from the exposed network traffic of the "honeypot" interface.

Because of the evolving nature of all products involved, these installation instructions will be fairly high-level. As mentioned earlier, QEMU is necessary for building and configuring the images to be used by Argos. Although some documentation suggests the necessity of installing KQEMU and QEMU concurrently, or including KQEMU within the QEMU directory when proceeding with the `./configure && make && make install` installation procedure, we didn't see that as necessary. These two packages were installed separately. The SDL library, which enables graphical functionality for QEMU and Argos, is necessary for obvious reasons.

The only caveat surrounding the install is to ensure that the SDL library is not already installed as part of the package distribution for your installed host system. If it is, there is no need to download the SDL library from source and install it. On a Fedora Core 9, if this package is installed previously and a user installs SDL from source, QEMU will work but Argos will throw an error, stating that SDL cannot be initialized. This can be a frustrating problem if one does not realize that two instances of SDL are installed. If encountering this problem, simply uninstall the "sourced install"; this should resolve the issue.

Finally, installing the Argos package simply requires the normal `./configure && make && make install` routine. A good place to install Argos is under /opt/argos. In addition, QEMU, KQEMU, and Argos require a gcc compiler version prior to 4.x. For research surrounding this article, gcc 3.4.6 was used.

Running Argos

Some network preparation must be done prior to launching an image in Argos. First the bridged interface must be set up. We brought this up on the eth0 interface and did not assign it an IP address. For security reasons, we felt that exposing eth0 to malicious traffic might provide an opportunity to compromise the host system: assigning no IP address resolves this issue. *Virtual Honeypots* [2], mentioned earlier, provides a good guide for bringing up the bridged interface.

The next step is to configure the argos-ifup script found in the Argos source package. We modified the script as follows to look a little different from what's provided from the Argos source:

```
#!bin/sh

sudo /usr/sbin/brctl addif br0 $1
sudo /sbin/ifconfig $1 0.0.0.0 promisc up
```

This script is used to bring up the network interface for the virtual machine when launching Argos. We also ended up writing a script to include all of the bridge setup commands. These are scripts that will need to be run over and over, so it's prudent to start setting up scripts to make that part of the process easier.

After all of this is done, Argos may be launched. Depending on your configuration, it may be necessary to start Argos using sudo, or the tap device will not initialize.

Strengths and Weaknesses

Argos works as advertised and is extremely useful in new and zero-day vulnerability research. The memory dumps provided when a system is compromised offer valuable information that researchers can use to understand an attack scenario and the exploit code in use. This infor-

mation can also aid in reverse-engineering the vulnerable process to understand the root cause in a timely manner. This is all achieved in a virtualized environment, which limits the management overhead.

Unfortunately, the memory dumps Argos provides are also a source of weakness for the honeypot. The memory dumps generated and subsequent .csi file outlining that memory for a particular attack could alone be several megabytes in size. The data is in a raw binary format; looking at it unaided, an analyst might not be able to make much sense of it. Furthermore, the Argos package does not include a .csi file parser to put this into an easily readable and usable format. Users are left to either comb through the file, which could take an unacceptable amount of time (think response and remediation for new vulnerabilities), or to write their own .csi file parser. A file-parsing library called cargos has been created and is available publicly; however, its output is still a bit raw and should be considered more of a demonstration of what cargos-lib can do. We created a separate .csi file-parsing utility that will be discussed later on in this article.

Another weakness of Argos, as with most attack fingerprinting and detection mechanisms, is false positives. As mentioned earlier, on at least one occasion, Windows update was observed to trigger Argos to log the event as an attack. This can be avoided by disabling the Windows Update mechanism (and, in general, disabling processes on images that might call for external input to the machine that could eventually result in a change of execution flow). Although this may be viewed by some as the product functioning as specified rather than a weakness, it is still an issue to consider when deploying honeypots with Argos.

Finally, we were able to produce a situation in which Argos failed to detect an attack. It is possible, in certain situations, to overflow data from one variable to another without affecting control flow information on the stack. Thus, an attacker can use this to rewrite the contents of another local variable. If a variable responsible for execution flow is affected, then the attacker can compromise the system. Argos cannot detect this, because it has no way of knowing the bounds of individual local variables—that would need compiler-dependent debugging information. It marks the initial variable as tainted and moves on. Figure 1 demonstrates a rather contrived scenario rarely seen in the wild; however, it illustrates a shortcoming of Argos.

```
#include <stdio.h>
#include <string.h>

int do_auth(void) {
    int good_password = 0;
    char pass[32];
    printf("enter password\n");
    gets(pass);
    if(strcmp(pass, "secret1") == 0) {
        good_password = 1;
    } else if(strcmp(pass, "secret2") == 0) {
        good_password = 1;
    }
    return good_password;
}

int main(int argc, char *argv[]) {

    if(do_auth()) {
        printf("password accepted\n");
    } else {
        printf("invalid password\n");
    }
   return 0;
}
```

FIGURE 1: EXAMPLE PROGRAM SHOWING A TYPE OF TAINTED VARIABLE EXPLOIT ARGOS CANNOT DETECT

In this example, it is possible to pass 33 or more characters to the password prompt without overwriting the return address of the `do_auth` function. This will set `good_password` to a nonzero value and thus make the program output "password accepted." Argos cannot possibly detect this because it has no way of knowing if `do_auth` has two local variables, one 32 bytes long and one 4 bytes long, or a single local variable that is 36 bytes long.

Additional Notes

As mentioned earlier, Argos also supports a feature in which it will inject shellcode into the exploited process to try to extract useful information; this feature is enabled by default. Currently, it only tries to extract the PID of the process. This information is sent to the local machine (the honeypot inside Argos) on port 8721. This means that you may want to have netcat or perhaps a quickly written tool constantly listening on this port in the VM to capture this information. Since the amount of information that could be gained is so large, we expect this to be expanded to provide even more information in the future. There are a few things we would like to see added in future versions, the most useful of which would be the process name and its full path. In addition to that, it would be very useful if the information were sent to the host machine and not to a port inside the VM, as this could be used by the attacker to detect that the targeted system is a honeypot.

Potential Deployments

Generally speaking, there are two deployment options for the Argos honeypot, and honeypots in general, that can be used. The honeypot can be set up either internally or externally. The deployment options are mostly self-explanatory but, to reinforce, an internal deployment would be exposed to local network traffic only, while an external deployment would be exposed to traffic in the wild. An internally deployed honeypot is the same as an externally deployed honeypot, since both will be able to examine current threats to a network, new vulnerabilities, and the behavior of malicious code that is traversing the network. However, the decision to deploy internally or externally depends on the intended use of the data collected by that honeypot.

An internal deployment will allow the honeypot user to see existing malicious traffic that might be traversing a local network. This gives a user insight into the threats already facing the network, as well as how malicious code might be traversing the network. Therefore, internal honeypot deployment is advantageous if the intention is to provide an added tool for understanding how to eradicate particular code from the network. An external deployment will allow the user to identify new threats, including new vulnerabilities and malicious code variants. So, while such a deployment can give a user the same information as an internal deployment, the goal is to identify new issues or variants of old issues to prepare network defenses and offset risk of compromise. Thus, an external honeypot is advantageous in keeping a proactive network security posture.

Argos is best deployed externally, as it is designed for new vulnerability discovery. Deploying internally and receiving information on a new vulnerability means one's network has already been compromised. In an optimal situation, deploying externally will provide information on a new vulnerability before it affects the internal network and thus provide the information necessary to offset exploitation risk presented by this new issue.

.csi File Parsing

The .csi files generated are in a relatively simple binary format. During the reviewing of Argos, we developed a utility to parse and display these files in a more useful way. The format of the .csi file is based on two main structures: the log header and zero or more memory blocks. The structures of each are detailed in Portokalidis, Slowinska, and Bos's paper, "Argos: An Emulator for Fingerprinting Zero-Day Attacks" [4].

Our parser is fairly simple to run. For the most part one will want to just run it with ./argos_parse argos.csi.1719845868 --tainted_only | less.

This will produce output like Figure 2, which was generated when exploiting a vulnerable program created for this research (and has had parts snipped for brevity).

```
NET TRACKER DATA:   false
HOST ENDIAN:        little-endian
HEADER VERSION:     2
TIMESTAMP:          Mon Sep  8 12:24:23 2008
GUEST ARCH:          x86
ALERT TYPE:         ARGOS_ALERT_CI

eax [0x00000000] (C ) ecx [0x0012f634] (C ) edx [0x7c9037d8] (C ) ebx [0x00000000] (C )
esp [0x0012f558] (C ) ebp [0x0012f56c] (C ) esi [0x00000000] (C ) edi [0x00000000] (C )
eip     [0x0012ffb0] (C )
old_eip [0x010111ac]
efl     [0x00000202]
---------------------------------
MEMORY BLOCK HAS NET TRACKER DATA:  false
MEMORY BLOCK VERSION:               1
MEMORY BLOCK TAINTED:               true
MEMORY BLOCK SIZE:                  4
MEMORY BLOCK PADDR:                 0x0e86f4fc
MEMORY BLOCK VADDR:                 0x0012f4fc
0012f4fc  aa 11 01 01               |....|
---------------------------------
MEMORY BLOCK HAS NET TRACKER DATA:  false
MEMORY BLOCK VERSION:               1
MEMORY BLOCK TAINTED:               true
MEMORY BLOCK SIZE:                  6e4
MEMORY BLOCK PADDR:                 0x0e86f91c
MEMORY BLOCK VADDR:                 0x0012f91c
POSSIBLE SHELLCODE BLOCK!
0012f91c  41  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 |AAAAAAAAAAAAAAAA|
...
0012fadc  41  41 41 41 41 41 41 41 41 41 41 41 eb 03 59 eb |AAAAAAAAAAAA..Y.|
0012faec  05  e8 f8  ff  ff  ff  4f  49 49 49 49 49 49 51  5a 56 |......OIIIIIIQZV|
0012fafc   54  58 36 33 30 56 58 34 41  30 42 36 48 48 30 42 |TX630VX4A0B6HH0B|
0012fb0c  33  30 42 43 56 58 32 42 44 42 48 34 41  32 41  44 |30BCVX2BDBH4A2AD|
0012fb1c  30  41  44 54 42 44 51  42 30 41  44 41  56 58 34 5a |0ADTBDQB0ADAVX4Z|
0012fb2c  38  42 44 4a 4f  4d 4e 4f  4a 4e 46 34 42 30 42 50 |8BDJOMNOJNF4B0BP|
```

FIGURE 2: EXAMPLE OUTPUT OF OUR OWN .CSI PARSER PROGRAM

As you can see, the program is able to successfully identify the "Possible Shellcode Block." The shellcode identified here, used when exploiting our vulnerable program, launches calc.exe and is borrowed from the Metasploit Payload Generator using the PexAlphaNum encoder [5]. Another value of interest, which is available as of .csi file version 2, is the old_eip field. In this example, the old_eip represents the address of the RET instructions that actually jumped to the tainted memory. Using other tools, such as IDA Pro, we can find the location of the vulnerable function (which is probably the one that ends with that RET); this information can be invaluable during analysis.

Though the cargos library does exist, we decided to implement our parser independently. The primary reason for this is that we wanted to spend the time to really understand how Argos works and the type of information it provides. Perhaps a future version of our parser will make use of cargos. As a minor footnote, we mention that although the tainted flag exists for each reg-

ister, on at least one occasion this flag was set to false when it should have been set to true. In other words, a tainted register was observed with a tag signaling that it was not tainted.

Conclusions and Future Work

Argos is an invaluable tool in the field of vulnerability research. Its ability to detect new vulnerabilities provides another tool for researchers tasked with vulnerability discovery and for network administrators responsible for network security. However, there is little application beyond new vulnerability discovery. Argos is not ideal for malicious code analysis and exploitation trending, since the only information collected is a memory dump when the flow of execution is altered by tainted registers or variables. Furthermore, Argos is still relatively new and is going through an evolution just like any other product in its infancy. Used alone, Argos produces output that requires a significant amount of time in order to discern any information of value. Programs that will be able to address this problem, such as cargos and the file parser we created internally, are just beginning to emerge. Although the malicious code problem could be argued to be a design choice, it is a topic to consider for functionality addition to Argos. A second suggestion for future work is to build a tool capable of listening on port 8721 for the PID of the compromised process, as per our "Additional Notes" section. Although more administrative in nature, aggregation functionality can also be considered to pool the data collected from several Argos deployments into a central location. This naturally leads to the idea of a central management interface to aid in the maintenance of running images deployed across different locations. For now, Argos is an excellent passive, high-interaction, virtual honeypot that is most useful to security research organizations and individuals with spare time and an interest in honeypots.

REFERENCES

[1] http://www.honeypots.net/.

[2] N. Provos and T. Holz, *Virtual Honeypots: From Botnet Tracking to Intrusion Detection* (Reading, MA: Addison-Wesley, 2007).

[3] http://bellard.org/qemu/about.html.

[4] http://www.cs.kuleuven.ac.be/conference/EuroSys2006/papers/p15-portokalidis.pdf.

[5] http://metasploit.com:55555/PAYLOADS?MODE=SELECT&MODULE=win32_exe.

[6] http://www.few.vu.nl/argos.