The Cyber Grand Challenge and the Future of Cyber-Autonomy

DAVID BRUMLEY

David Brumley is the CEO and co-founder of ForAllSecure, and a Professor at Carnegie Mellon University in ECE and CS. ForAllSecure's mission is to make the world's software safe, and they develop automated techniques to find and repair exploitable bugs to make this happen. Brumley's honors include a United States Presidential Early Career Award for Scientists and Engineers (PECASE), a Sloan Foundation award, numerous best paper awards, and advising one of the world's most elite competitive hacking teams.
dbrumley@forallsecure.com

The Cyber Grand Challenge was about much more than a capture-the-flag (CTF) competition between computers. The people who built those systems had to learn how to replicate the behavior of human hackers, perform binary program analysis, patch vulnerable applications—or not, if installing the patch hurt performance or resulted in a functional regression. In this article, based on my 2018 Enigma talk [1], I will describe the competition, and also how human hackers and the CGC competitors' systems have different strengths.

I want to begin by introducing you to the person whom I believe is one of the world's best hackers. His name is Loki and he's an expert at web browser security. At the 2016 Pwn2Own competition [2], Loki demonstrated three new vulnerabilities and was able to exploit them in applications that would have enabled him to break into 85% of the computers in the world.

The rules for Pwn2Own are actually pretty simple. The people running the contest install a fully patched version of an operating system on a laptop, and then they install the latest, greatest, direct-from-developers version of a web browser. The goal is to break into the computer through the web browser. If you think about it, this is pretty amazing because vendors spent a lot of money trying to secure their operating systems. But Loki has been studying computer security, is an expert in the internals of Google Chrome, and has been studying web browsers for a long time.

Loki sat down in front of a laptop running Google Chrome on top of Windows and within two minutes had demonstrated a vulnerability. What's amazing is that over the course of the next two days he also demonstrated new zero days in Microsoft Edge and Apple Safari. Those were the three vulnerabilties that would have allowed him to break into 85% of the world's computers.

Loki is the world's best hacker, in my opinion, but he's not a criminal. I don't want to conflate the terms hacker and criminal. Loki responsibly disclosed those vulnerabilities to vendors, and those vendors issued updates that protected millions of people.

Over the course of that weekend Loki also made $145,000. Not bad for 15 minutes of work, keeping in mind that like a professional athlete, Loki spent a lot of time preparing for this contest.

Software and Vulnerabilities

Think of all the software that you use every day. And I'm not just talking about the software that's running on your computer, your laptop, or on your smartphone. I'm also thinking about all this software that you interact with on nonobvious devices: IoT devices, your WiFi router, even the software that powers the safety system on your car. Who's checking it for security vulnerabilities?

How do we go about doing what Loki does and do it at scale? On just the Google Play and Apple stores, a new app is released every 13 seconds. How do we go about checking software when a new app is released so frequently? I've been working on this problem for a long time, along with other academic researchers. If we could take what Loki does and program computers to emulate it, computers could do that work for us.

At Carnegie Mellon University, we built a tool called Mayhem that takes off-the-shelf software and audits it for vulnerabilities. As an example, we used Mayhem to explore `iwconfig`. Mayhem systematically explored the state space in `iwconfig` and output an exploit. You can take that exploit, give it as input to `iwconfig`, and get a root shell. At a basic level, we enabled a computer to take a software binary, autonomously find a security vulnerability, and prove it with a working exploit. This isn't the world that we live in today, but I think this is the world that we need to live in—one where not just developers can check for the security of applications but anyone can, using systems like Mayhem.

Scale

In one study we used Mayhem to examine 37,391 programs—every Debian program available. We spent three years of CPU time analyzing the programs: that amounts to five minutes per application. We did this in an embarrassingly parallel way by bringing up a bunch of Amazon nodes. We found 2.6 million new crashes due to 13,875 new bugs in those programs. Of those, 250 exploits would allow getting a shell.

We want to be operating at the scale where we can check the world's software for exploitable bugs. Doing analysis like this cost us 28 cents per new bug and $21 per exploit. Compare that to Loki who made $145,000 for three working exploits.

Cyber Grand Challenge

We've been doing this research at CMU for over a decade, along with other researchers such as Giovanni Vigna at UC Santa Barbara, Dawn Song at UC Berkeley, and a much larger community. DARPA, the Defense Advanced Research Projects Agency, stepped up to challenge this community in an open forum. DARPA issued the Cyber Grand Challenge, wanting to do for cyber what the autonomous driving challenge [3] did for vehicles. DARPA wanted to turn cyber into something that was completely autonomous and controlled by computers. They challenged the community to combine the speed and scale of automation with reasoning abilities that exceeded those of human experts.

DARPA first issued an open call for proposals for a fully autonomous offense and defense contest, with a $2 million cash prize for the winner. That got lots of attention, and over 100 US entities registered for the CGC. At the end of the first year, DARPA pared down the entrants based on the same performance factors to be used in the final contest, leaving seven contestants.

The final contest occurred at DefCon 2016. This contest was different from the usual Capture the Flag (CTF), held at DefCon every year, which pits human teams against one another. In the CGC, it was computer against computer, with an air gap separating the computers from any attempt at outside help or interference.

DARPA structured the event by sending programs to all the contestants' systems. These systems needed to find vulnerabilities in those programs and create patches that fixed those vulnerabilities, sending patches back to the DARPA moderator. The DARPA moderator system evaluated the security solutions based on a number of factors. The first was if you created an exploit, does it work against other people's patched binaries. DARPA called this "consensus evaluation": you get points based on whether your exploit works against other people's running programs—patched or not.

Second, the patch itself is evaluated for security, checking to see whether the patch itself could be exploited.

But the world isn't just about security. It's also about things like functionality. DARPA would take the patched binaries and make sure that the system retained all its original functionality. After all, if a patch breaks your system you're not going to install it. They also measured the performance of the patch. DARPA's moderator would look at things like memory overhead.

And so when you considered the type of contest, it wasn't all about security. It was about making decisions that operated within a confined space to make sure that it wasn't just the most secure but also maintained performance and functionality. To give you an idea, on our system, if we determined that our patches had more than 5% overhead, it may have been better to play the original buggy binary.

Round after Round

When a competitor found an exploitable bug and submitted a patch, things didn't end there, because that's not how the world works. DARPA changed the direction of the community by saying the goal here is to win, and the way you win is by giving people access to your patches. DARPA's moderator would take our patches and give them to our competitors, and the competitors could do further analysis to see whether they could circumvent them. They could try to steal our patches and use those patch techniques themselves and submit them back to the DARPA moderator round after round. So the CGC wasn't just about security and a point in time, but about security as it evolves. It allowed attackers and defenders to learn from each other. And by the end of the contest, we'd completed over 95 rounds to determine a winner.

When we looked at the scope of CGC, we had to do three things. First, we needed to be able to automatically exploit software. We had to do what Loki does to find vulnerabilities, and we had to teach a computer to do it. Second, we had to be able to automatically rewrite binaries to add in defenses to prevent them from being exploited again. And third, just as importantly, we had to make better decisions than our opponents. That was huge.

Automatic Vulnerability Discovery

Let's talk about how we went about doing the automatic vulnerability discovery. We needed to be able to perform code analysis without source code. We built the binary analysis platform (BAP, [4]) at CMU, available for free from GitHub. BAP allows us to take a binary and raise it up to an intermediate representation that is useful for doing program analysis.

We then created tools to find vulnerabilities in software using a technique called "symbolic execution." We, and others in the community, considered symbolic execution as a very academically promising technique that we could publish papers about while advancing the frontiers of research. But we discovered during this contest that trying to find the best single solution was the wrong thing to do.

Instead, we realized that applying a portfolio of techniques, such as fuzzing and crash exploration, would be more effective. If you have a crash, the program has some sort of mental problem with the world at that point. The program *thinks* the world is different than it actually is; if there's a bug in one place, there's likely a bug nearby as well. We also built a feedback loop between these techniques which allowed us to find far more vulnerabilities than any single technique alone.

At the end of the contest we found 67% of our bugs with fuzzing and 33% with symbolic execution. But those percentages only reflect the final uncovering of the bugs themselves. We found that the symbolic executer would often reach a promising part of a code then hand that over to a fuzzer, which used a different set of techniques. It would be the fuzzer that ultimately found the vulnerability, but only after being enabled by the symbolic execution technique. We learned that building a portfolio of techniques that work together cooperatively is always going to outperform any single technique.

Defense

For defense, we had to be able to statically rewrite binaries. We used data flow analysis as a basis to focus formal program analysis on understanding how a program worked. We could then derive an analysis that would rewrite the program. Here too we used the portfolio, with two techniques. The first we called hardening, rewriting the binary to essentially introduce seatbelts: control flow integrity, stack canaries, ASLR, DEP, and so on. Now these remediations are agnostic about whether there is a vulnerability or not, but they make the program safer overall. Second, when we found a particular bug, we'd automatically rewrite that portion of the code where the bug occurred to introduce safety checks.

After hardening the binary, we could add crash-specific patches for vulnerabilities when we found them. For example, one of the challenges DARPA gave us in CGC was based on the SQL Slammer worm. When Mayhem receives a binary, it immediately starts generating automatic regression tests, which function as the baseline for the binary. We'd start patching, creating a hardened binary and then replaying those test cases to make sure that we had no loss of performance and functionality. We would also go in and rewrite the patches, replaying those automatically generated test cases to measure and ensure that functionality and performance were maintained. Then we would have to decide which of these patches to apply. Instead of having just one patch, we had an array of patches based on the portfolio of techniques. We would measure them and empirically determine the best ones to deploy at a particular time. Finally, we had to build a system that was completely autonomous.

Dynamic Scaling

When DARPA started the contest, we didn't know whether they were going to give us 10 programs, 100 programs, or 1000 programs. So we had to build in the capability to dynamically scale our environment to dedicate resources where they were going to matter most. For example, if we have a program that we keep finding new bugs in, it would make sense to devote more resources to that than to a program that's not buggy.

Second, we had to make sure that we were playing the game the optimal way. For example, if we created a patch and that patch had 5% overhead, we might decide that it was too dangerous to play since that 5% overhead would hurt our score unless we thought someone was attacking us.

In a nutshell, we would do the binary analysis, we'd harden, we'd find exploits, we'd patch, and we'd run through a decision process that determined the best security solution. Then we'd deploy and iterate through this process again and again and again.

In the CGC, we faced some of the most notable names in program analysis out there: UC Berkeley and UC Santa Barbara, who have been doing research on this forever; Raytheon, a large defense contractor, was also in the final seven contestants. We also had a two-person team from the University of Idaho who qualified, beating out 93 other teams.

At the end of this contest, Mayhem won. ForAllSecure, the company we founded to continue the development, got the $2 million cash prize. But when you looked at the contest, everyone had little twists on their techniques that were different, and if you look at the scores you'll notice they're not very far apart.

CTF

At the end of the CGC, our system, Mayhem, got to participate in the annual DefCon 24 CTF [5] (Table 1 shows the final results). Just to give you an idea of the caliber of the people Mayhem played against, the number three team DEFKOR had Loki on it, the person who demonstrated three zero-days over the course of two days.

Team	Score
PPP	113555
b1o0p	98891
DEFKOR	97468
HITCON	93539
KaisHack GoN	91331
LC⚡BC	84412
Eat Sleep Pwn Repeat	80859
Binja	80812
Pasten	78518
Shellphish	78044
9447	77722
Dragon Sector	75320
!SpamAndHex	73993
侍	73368
Mayhem	72047

Table 1: DefCon 24 CTF results [5]

We had built Mayhem as part of ForAllSecure, and I've also been the faculty mentor of the human hacking team at CMU, PPP (Plaid Parliament of Pwning). So I have some insight into what the machines could do versus what the humans could do. Although the machine lost, you'll notice from the scores that it was competitive. For two out of the three days of the DefCon 24 CTF, the machine was beating some of the teams.

But there are differences. For example, if you look at the humans, they have an incredible notion of being able to abstract details (Table 2). It's something that humans are great at, while the machine has precision going for it. At one point in the contest, for example, aPPP was looking at a line of code they thought might be vulnerable but couldn't figure out how to exploit. Mayhem was able to create an exploit because it had to reason about program branches and the particular program state, which was extremely complicated and would far exceed human understanding. Mayhem was able to do that in a fraction of a second.

Second, humans have intuition, and this is extremely important when you hack because you have to decide where to focus your attention. You may think, this part of the code looks extremely tricky, and therefore I'm going to focus my attention there.

Machines have brute force. Very simply, brute force is incredibly useful when you're trying to analyze, exploit, and patch applications at scale. And the way we see it from our research point of view is that once a person has an intuition of where to look, brute force can be used as a leverage point.

Human	Machine
Abstraction	Precision
Intuition	Brute force
Creativity	Scalability

Table 2: Human vs. machine qualities when it comes to hacking, as well as other forms of problem solving

Finally, humans have creativity. The machine will only look for vulnerabilities that it has been programmed to look for, while humans aren't restricted. Attackers and defenders get to co-evolve. For example, there is a class of attacks called timing attacks. The way I describe them is as follows: suppose my wife asked me, "Do I look fat in these pants?" and I took a few seconds to respond. It doesn't really matter what my response is now. The amount of time it took me to respond reveals all the information needed. It's the same way in security, where the amount of time it takes to do something can reveal something about the secret.

Humans have this great creativity to invent new classes of attacks. While the machine, once we program it to look for that class of attacks, has enormous scalability in looking at all the programs in the world.

Conclusion

In this article, there have really been two themes. The first theme is that human effort alone does not scale. Apps are being released at a pace that far outstrips people's ability to examine every one. Yet we need something as a safety checkpoint to make sure we've looked at every piece of software for security vulnerabilities. It's just too important not to do.

Second, we can teach computers to hack. Humans can teach computers to do at least a little bit of what Loki does and apply that to every piece of software in the world.

References

[1] Enigma 2018 talk: https://www.usenix.org/conference/enigma2018/presentation/brumley.

[2] A. Armstrong, Pwn2Own 2016—The Results: http://www.i-programmer.info/news/149-security/9556-pwn2own2016.html.

[3] https://www.darpa.mil/about-us/timeline/-grand-challenge-for-autonomous-vehicles.

[4] Binary Analysis Platform: https://github.com/BinaryAnalysisPlatform/bap/graphs.

[5] DefCon 24 CTF: https://www.defcon.org/html/defcon-24/dc-24-ctf.html.