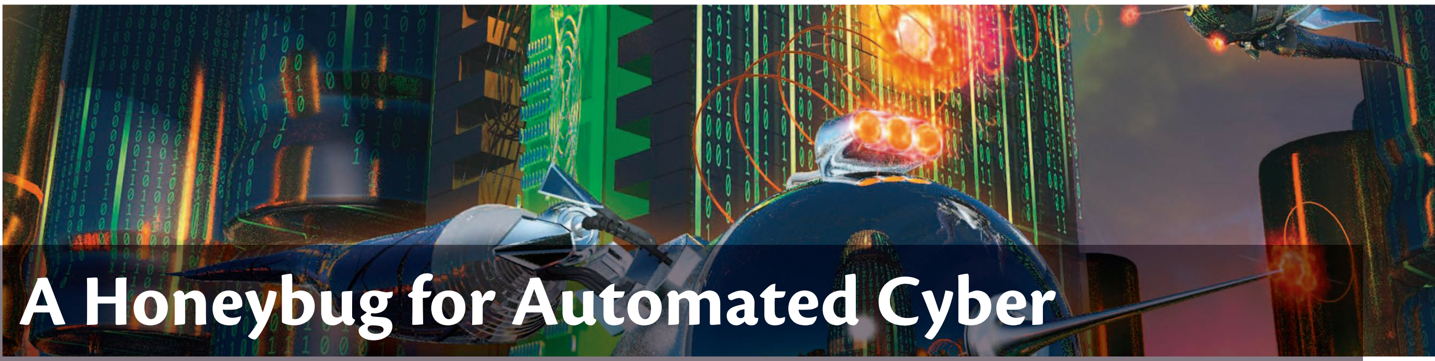

A Honeybug for Automated Cyber Reasoning Systems

Timothy Bryant and Shaun Davenport | Raytheon

From the editors: While all the articles in this issue share the special issue theme, two are uniquely related in that they present differing perspectives of one particular strategy employed in the Cyber Grand Challenge Final Event. In "A Honeybug for Automated Cyber Reasoning Systems" the authors (creators of the Rubeus system) detail an approach aimed to thwart the anticipated analysis techniques of other competitors. Contrarily, "Effects of a Honeypot on the Cyber Grand Challenge Final Event" provides an analysis of this very approach from the perspective of a contest referee—a member of the competition framework integrity team. Between the two perspectives, one sees the approach's reasoning and potential risk as well as an unbiased analysis of the implementation and an account of its efficacy on the other competitors.

In the Cyber Grand Challenge, our cyber reasoning system patched binaries with a honeybug that enticed competitor systems to pursue the honeybug instead of exploiting actual vulnerabilities. Methods for detecting and countering automation in the real world may be used to thwart the ability of malicious actors to find vulnerabilities.

"The enemy knows the system" was an essential design principle for DARPA's Cyber Grand Challenge. Also known as Shannon's Maxim,[1] it ensured that competing automated cyber reasoning systems (CRSs) could evaluate the effectiveness of each other's defenses. A cyber reasoning system automates the search for vulnerabilities and applies appropriate patches to protect the software from future attacks. In the Cyber Grand Challenge, CRSs did not analyze black boxes across a network, but instead analyzed competitor binaries disclosed to them by the game infrastructure.

A honeybug is a pseudo-vulnerability planted in a binary with the goal of diverting the attention of a CRS away from legitimate existing vulnerabilities. Our honeybug patch was a strategy used by Rubeus, our CRS, to subvert our enemies' ability to "know the system." Although we could not prevent the distribution of our binaries, we could perhaps trick competitors if our binaries behaved differently on their systems than they did on the official scoring system. If so, it would be possible to prevent competitor systems from searching for legitimate proofs of vulnerabilities (PoVs) in our binaries or prevent them from validating legitimate PoVs against our binaries.

Moreover, if our binaries behaved differently on competitor systems, we could manipulate those systems because they would be interacting with data that we controlled (that is, our patched binaries). Detecting emulated or virtualized environments is a well-studied problem and various techniques have been developed[2]—and human teams playing capture-the-flag (CTF) games have used shenanigans to gain an advantage over other teams.[3] In contrast, the intent of our honeybug was to build into our CRS the ability to detect other cyber reasoning systems and to manipulate them.

Enabling Shenanigans

Ironically, this strategy was made possible because the designers of the Cyber Grand Challenge afforded themselves the dubious benefit of "security through obscurity" by using an undisclosed platform for scoring. While our binaries were to be fully exposed to competitors, the teams themselves had little visibility into the execution environment of the scoring system.

The DECREE (DARPA Experimental Cybersecurity Research Evaluation Environment) operating system (https://github.com/CyberGrandChallenge), with its carefully circumscribed set of seven system calls, removed the ability to securely bind the execution of our binaries to our designated host. In addition, DECREE prohibited high-fidelity timing information (for instance, RDTSC instruction), which can often be used to detect different execution environments. Thus, for many months we assumed there was no mechanism available to our binaries for detecting whether or not they were running on the official scoring system.

A couple months before the final event, we considered the CPUID instruction that reports details about the underlying processor. We could use it to identify the official scoring system, but only if the processor used by the scoring system was both unique and generally unknown to the rest of the competition. Otherwise, teams would certainly update their CRS's model of the execution environment to match that of the official scoring system. In addition, we recognized that because our CRS was allowed to monitor the network traffic to and from our binaries running on the scoring system, we could exfiltrate the CPUID values from our binaries back to our CRS by embedding those values in the data that our binaries transmitted.

A second enabler of our strategy was that our competitors were fully autonomous machines. This meant that no humans would be scrutinizing the CPUID check, recognizing its purpose, and removing the check from our binary. To avoid falling for this trick, a CRS must understand the purpose of the code—that normal functioning occurs on the official scoring system (or a system with identical CPUID values) and that alternative code is executed otherwise. Unless the other teams had preprogrammed their CRSs to recognize and react to the CPUID check (or they had correctly modeled the scoring system's execution environment), there was a strong chance that their CRSs would fall for the trick.

Fingerprinting the Scoring System

Because this technique was predicated on secrecy, we wanted to quietly fingerprint the scoring system. However, we didn't have access to the scoring system until the testing began just a few weeks prior to the final event. During the testing phase, our CRS competed against mock sparring partners to verify that our system was operational and ready for the final event. If, during these sparring sessions, the CGC organizers discovered that we were using the CPUID, they could decide to change the values reported by the scoring system before or even during the final event. Alternatively, they might publish the CPUID values and guarantee to all teams that those values would remain fixed. Such actions would prompt our competitors to scramble to update their CRS to properly model this aspect of the scoring system.

So, we submitted a single patched binary to the scoring system during one of the test runs. Instead of submitting a correctly secured binary, the purpose of this special binary was to query a large range of CPUID values and to transmit the values as data over the network to be picked up by our CRS that was monitoring the network traffic. As the day of the competition drew near and the CGC team indicated that the scoring system code was tested and "frozen," we worried less about being discovered, and our exfiltration of the CPUID values from the scoring system even included the playful plaintext phrase "NOT FOR YOUR EYES."

"The intent of our honeybug was to build into our CRS the ability to detect other cyber reasoning systems and to manipulate them."

We later learned that the infrastructure team observed our fingerprinting activities just a few days before the final event during a monitored test run, and it prompted the team to fix a bug in their visualization system. They would soon be projecting their visualizations of running programs on large screens in front of a live audience. To perform these visualizations, this part of their system replicated the execution of competitor binaries in an emulated environment. However, that environment was not identical to the scoring system. In particular, it reported different CPUID values. Thus, they had to fix the CPUID values being reported by the emulator. Otherwise, the audience would have seen the honeybug execution path instead of the actual execution occurring on the scoring system.

The fingerprinting results were quite perplexing. The scoring system claimed it was using an Intel i5-4258U processor—a common processor for laptops, not a server with thousands of CPU cores. The scoring system environment was reporting additional CPUID values outside the range of what Intel x86 processors are supposed to report. Moreover, it was reporting values that were different from commonly used emulators and hypervisors. We were able to conclude that the scoring system was either heavily modified or an entirely custom emulator or hypervisor. We settled on 12 CPUID values that best differentiated the scoring system from other emulators and hypervisors that were likely to be used by our competitors.

Of course, we had to ensure that our CRS correctly modeled the CPUID values of the scoring system. Otherwise, it would be susceptible to the very trickery we hoped to use against our competitors. During the final event, the fingerprinting patch checked if the current CPUID values reported from the scoring system had changed. If the values had indeed changed, the patch would sacrifice our availability score to report back the correct values (that is, the patched binary could either exfiltrate values from the scoring system or respond correctly to the functionality tests, but not both at the same time). Thus, instead of transmitting the proper response, our binary transmitted the CPUID values. Because our CRS monitored the network traffic to and from our running binaries, it would pick up the new values and distribute them to the rest of the system, updating our hypervisors and patching subsystem accordingly. Although this never occurred during the final event, our CRS was prepared.

Also, if the reported values changed too often (we used a conservative threshold of more than 1 for the final event), our CRS would cancel the whole scheme. Not only would it not be able to employ any shenanigans, but neither would any other CRS. Therefore, it would no longer need to update its model of the scoring system.

Choosing the Effect

Now that our binary could detect when it was running in a competitor CRS's environment (that is, anywhere but the scoring system that we had fingerprinted), we wanted to leverage this capability to our greatest advantage. We considered various options including denial of service, evading analysis, and remote code execution.

At the time, we did not consider a Rowhammer-style attack[4] to disable competitor systems. This style of attack manipulates DRAM memory cells to corrupt memory regions that are otherwise protected by the operating system against unauthorized access. Because all of the CRSs ran on identical hardware and we knew the exact hardware specifications, and because we could also reasonably guess the hypervisors and emulators used by our competitors, this attack may have been feasible. However, disabling a competitor's CRS using this method would have violated the official rules and participation agreement.[5,6]

One idea that we considered was making the patched program run in a computationally intensive infinite loop. Although it would be beneficial to consume competitors' analysis resources, we figured that most CRS implementations would enforce timeouts and that it would probably not have a huge impact on systems with 1,024 CPU cores. We also considered using a simple terminate syscall—prematurely ending execution to evade dynamic analysis. Although this may prevent a vulnerable CRS from locally validating a legitimate PoV against our binary, PoVs that were already submitted would continue to be used for scoring unless they were explicitly replaced. Consequently, we turned our attention to manipulating competitor CRSs into unwittingly submitting invalid PoVs.

"We made the bug so obvious and inescapable that only a machine would not stop to ask why it was there."

Our honeybug was a planted vulnerability that we wanted CRSs to easily find and prove. Specifically, we implemented a simple stack buffer overflow that only required 12 bytes of input to overwrite the return address and thereby gain code execution. We also ensured CRSs could readily control both the program counter and a general-purpose register (control of both was required for a Type 1 PoV). We made the bug so obvious and inescapable that only a machine would not stop to ask why it was there. We hoped, however, that CRSs were sophisticated enough to actually go about the business of analyzing our patched binary.

The mechanics of patching binaries with the honeybug were straightforward. Our CRS deployed the honeybug opportunistically—only when it was already patching a bug or adding a defensive mitigation—because the penalty for patching was high. The CRS also ensured that the honeybug patch would always execute before our other defensive patches such as address randomization (ASLR) or nonexecutable stack (NX).

Gambling with a Honeybug in Vegas

Planting a honeybug in our binaries was not without risk. Backdoors that rely on secrecy are often discovered and reduce security. Although our honeybug is not a traditional backdoor, it could make some vulnerabilities easier to prove. For example, suppose a CRS found an existing vulnerability where it controlled the program counter (`eip`) but not a general-purpose register. Normally, such a bug would not be exploitable. However, if we insert the honeybug, an intelligent adversary can then exploit the original bug by simply redirecting code execution to our honeybug, thereby gaining control of `eip` and a general-purpose register and proving the vulnerability that we introduced! If our CRS were competing against humans, we would not choose to use a honeybug because skilled human CTF players would not fall for the CPUID trick and they would readily take advantage of the planted vulnerability.

Against machines, we gambled that the designers of the machines had presupposed that reference (unpatched) binaries are always easier to exploit than patched binaries. The reasoning is that patched binaries are likely to include anti-analysis and exploitation mitigations. Even if some patches are not completely effective, they would be unlikely to intentionally introduce new vulnerabilities. Therefore, we expected that CRSs would not look for novel vulnerabilities introduced by our patches.

Furthermore, in a multiplayer game with several CRSs issuing patches, there could be many versions of the same binary to analyze. For efficiency, we expected CRSs to allocate their computational resources to finding vulnerabilities in the reference binaries and only afterward adapt those PoVs to patched binaries.

Honeybug Enticement

We also wagered that CRSs would behave systematically, indifferently, and greedily. The common expectation was that challenge binaries would contain few vulnerabilities, perhaps even just one. We believed competitor systems would be designed to readily act on any bug discovered. Thus, if the honeybug forced a program to crash at the very beginning of its execution, a CRS would detect this crash, construct a corresponding PoV, and submit the new bogus PoV.

Our honeybug targeted machines that analyzed our binary or machines that actively verified and adapted PoVs that they had already discovered in the unpatched binary. We didn't expect the technique to work against all CRSs. If it worked against all CRSs, it would be a simple and complete defense. To increase our security score, the honeybug would need to entice all CRSs that were currently throwing a valid PoV. Although this scenario was unlikely, we did expect to manipulate some of the CRSs and decrease their evaluation score.

Interestingly, a less sophisticated CRS is less susceptible to this trickery. For example, a simple CRS may submit PoVs that it finds against a reference binary without first validating that they work against patched binaries. Another example is a CRS that does not or cannot adapt PoVs it finds in reference binaries to work against our honeybug patched binary. Because the CRS does not submit a different PoV, the scoring system continues to use the previously submitted PoV. Thus, this CRS is unaffected by the honeybug.

Table 1. Cyber reasoning systems submitting PoVs to exploit the honeybug.

Cyber reasoning system	# of honeybug PoVs scored	# of unique challenges
Jima	96	27
Mayhem	65	7
Galactica	31	4
Mechaphish	9	2
Crspy	2	1
Xandra	0	0

Results

We examined the results[7] from the Cyber Grand Challenge final event to determine how frequently competitor CRSs attempted to exploit the honeybug. It is difficult to assess the extent to which our honeybug impacted our final placement, because without knowing the internals of competitor CRSs, we can only speculate about possible adverse impacts to their overall performance. Moreover, the potential of the honeybug to alter overall scores depends in part on how many challenges for which CRSs are able to find proofs of vulnerability. Nevertheless, we were able, after the competition, to evaluate each PoV that was submitted against our patched binaries containing the honeybug. For each round where this occurred, we ran the competitor's PoV 10 times using random seeds. We configured our test environment to report CPUID values that differed from the official scoring system to force execution to the honeybug. Because the honeybug is vastly different from the real vulnerabilities in the CGC corpus, a successful PoV in our test implies that the honeybug was being targeted by the competitor. We ran this test for all rounds of the final event and tallied the results in Table 1.

Beyond the Cyber Grand Challenge

Is our honeybug a mere curiosity, taking advantage of a unique discrepancy between the Cyber Grand Challenge scoring system and the automated cyber reasoning systems? Or, does it actually point to something more relevant to the security community? Our honeybug illustrates that automated systems may need to reason about whether or not software is acting deceptively. It also clearly shows that cyber reasoning systems may be susceptible to manipulation. For CGC, the honeybug detected competitor CRSs by proxy using the execution environment. Perhaps, however, detecting the execution environment is not the only way to detect and manipulate a CRS.

Let's consider one important example. Today, most vulnerabilities in binaries are still discovered by fuzzing tools (many of which can be considered a primitive CRS). In fact, many CRSs rely heavily on fuzzing to produce a majority of the vulnerabilities they find,[8–10] and yet we are unaware of any software today that intentionally thwarts fuzzing-based tools. Fuzzing typically requires executing millions of test cases before a vulnerability is discovered, and many systems targeted by malicious actors are fuzzed without resetting the state of the system between each test case. This happens when there is no suitable emulator for the system, or because there is no ability to rapidly reset the system to a clean state between test cases.

Software that is permitted to retain state while being fuzzed (a condition that was not satisfied for the Cyber Grand Challenge) could differentiate between a fuzzing campaign and legitimate use. Password-based authentication mechanisms have long used the strategy: too many illegitimate attempts and either you lock yourself out or a significant delay is imposed on subsequent attempts. In a similar manner, one can imagine software that detects a fuzzing campaign and that imposes delays or redirects execution to dead ends or false bugs.

The mechanism for detecting and diverting a fuzzing campaign could be made difficult for humans to reverse engineer, let alone automated reasoning systems. Devising an efficient and effective defensive mechanism of this sort would be an interesting line of research. CRSs could continue to be used by software vendors in their development processes to find vulnerabilities before the software is shipped by using builds that do not contain the protections. However, the shipped software with the protections in place would reduce the ability of others to use CRSs to find vulnerabilities for malicious purposes, at least for some categories of software products.

Our honeybug demonstrates that cyber reasoning systems, like other software, may be susceptible to manipulation. In our example, the detection mechanism leveraged a difference between a CRS's model of the execution environment and the real execution environment. Once detected, our honeybug was able in some cases to prevent further analysis of our binaries and to provoke competitor systems to submit bogus proofs of vulnerability. The honeybug was a small but intriguing component of the Rubeus cyber reasoning system. ■

References

1. C. Shannon, "Communication Theory of Secrecy Systems," *Bell System Technical Journal*, vol. 28, no. 4, Oct. 1949.
2. P. Ferrie, *Attacks on Virtual Machine Emulators*, white paper, Symantec Corporation, Jan. 2007; http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf.
3. B. Schmidt and P. Makowski, "Dissecting Wireshark: A Case Study on Network Anti-Forensics," OSDFCon, 2014; https://www.osdfcon.org/presentations/2014/Schmidt-OSDFCon2014.pdf.
4. Y. Kim et al., "Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors," *Proc. 41st Int'l Symp. Computer Architecture* (ISCA 14), 2014; doi:10.1109/ISCA.2014.6853210.
5. "CGC Frequently Asked Questions," DARPA, Aug. 2016; http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite/Files/CGC_FAQ.pdf.
6. "CGC Participation Agreement," DARPA, 16 May 2014; http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite/Files/CGC_Extended_Application_form_v5_16_May_2014.docx.
7. B. Caswell, "Cyber Grand Challenge Corpus," 4 Jan. 2017; http://lungetech.com/cgc-corpus.
8. S.K. Cha et al., "Unleashing Mayhem on Binary Code," *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.
9. N. Stephens et al., "Driller: Augmenting Fuzzing through Selective Symbolic Execution," NDSS, 2016.
10. "American Fuzzy Lop," coredump; http://lcamtuf.coredump.cx/afl.

Timothy Bryant is a principal engineer at Raytheon developing automated vulnerability assessment technology. He led Raytheon's Deep Red team in the Cyber Grand Challenge. His research interests include dynamic and static analysis of binaries and artificial intelligence applied to computer security. Bryant has an MS in computer science from Columbia University. Contact at timothy.k.bryant@icloud.com.

Shaun Davenport is a vulnerability researcher at Raytheon. His interests include static analysis and embedded device vulnerabilities. Davenport studied computer science at Florida Institute of Technology. Contact at shaungdavenport@gmail.com.