Mechanical Phish: Resilient Autonomous Hacking

Yan Shoshitaishvili | Arizona State University
Antonio Bianchi and Kevin Borgolte | University of California at Santa Barbara
Amat Cama | Independent Researcher
Jacopo Corbetta | Independent Researcher
Francesco Disperati | PayJunction
Audrey Dutcher | University of California at Santa Barbara
John Grosen | Massachusetts Institute of Technology
Paul Grosen, Aravind Machiry, and Chris Salls | University of California at Santa Barbara
Nick Stephens | Independent Researcher
Ruoyu Wang and Giovanni Vigna | University of California at Santa Barbara

Recently, the vulnerability analysis process has started to shift from human analysts to automated approaches. The DARPA Cyber Grand Challenge featured cyber reasoning systems, such as our Mechanical Phish, that analyze code to find vulnerabilities, generate exploits to prove the existence of these vulnerabilities, and patch the vulnerable software.

Our world is becoming increasingly connected, and the fantastical view of hackers, as portrayed by *Hackers* and other '90s-era movies, is starting to seem feasible, but with nation-states and criminal enterprises taking the place of Angelina Jolie and her crew. Because we have repeatedly demonstrated the lack of sufficient collective security experience (or sufficient interest in software security) to avoid widespread vulnerabilities, research has turned to the automatic discovery and repair of such flaws in deployed software.

One driver of this research direction is DARPA, who has a long track record of pushing for the automation of tasks traditionally (and imperfectly) handled by humans. DARPA bootstraps research areas through a time-proven method: explicit competitions. To advance autonomous security analysis, DARPA organized the Cyber Grand Challenge (CGC), a competition in which human teams built fully autonomous cyber reasoning systems (CRSs) that were pitted against one another in a contest to analyze, exploit, and patch binary software.

Like DARPA's earlier self-driving Grand Challenge, the CGC was a proxy for a realistic scenario. The first self-driving Grand Challenge was held in the desert, and the resulting prototypes would suffer driving in a city as much as the prototype CRSs that came out of the CGC would suffer in the analysis of truly real-world software. But these systems represented a start: the CGC revealed a glimpse of a possible future in which machines not only build our cars, drive us around, and manage our homes but also ensure the security and reliability of the software we use every day.

 Copublished by the IEEE Computer and Reliability Societies

We have discussed our CRS, Mechanical Phish, from a technical perspective in literature[1] and in a number of conference talks. In this article, we not only provide these technical details but also discuss the human side and organizational side of the creation of a CRS and the lessons that the CGC taught us about cyber autonomy.

The Cyber Grand Challenge

Traditionally, groups of humans faced off in capture-the-flag (CTF) competitions designed to push their hacking skills to the limit. In these competitions, each group is responsible for the defense of a networked computer. Because the computers defended by the teams have the same configuration and installed services, each team works on finding vulnerabilities in their instance, and then use the acquired knowledge to fix the found vulnerabilities—and, at the same time, break into the computers run by the other teams. Each successful hack produces a secret "flag," which is presented to the organizers of the competition to prove that the service has been compromised. Although it started as an event for pure enthusiasts, CTF competitions quickly evolved into something resembling more of an e-sport, with longstanding, well-known teams, corporate sponsorship, significant media coverage, and the occasional scandal or novel development to shake up the field.

The CGC was one such development. In the CGC, DARPA created a nearly traditional competition with one fundamental twist: no humans could take part. Instead, participants had to create a system that could reason about cybersecurity in a fully autonomous way. The idea was that these CRSs would face each other in a competition in which the human factor was completely removed, and only automated approaches that were able to deal with the complete identification-patch-exploitation pipeline could be used.

Feasibility Concerns

There are many challenges that must be surmounted when developing a CRS. Some of these—the prioritization of paths during symbolic execution, the improvement of precision during static analysis, and so on—require as-yet unknown scientific advancements to be solved. Others seem to be mostly engineering challenges, simply requiring a large development effort by many skilled developers.

One of the biggest engineering challenges facing CRSs is environment modeling. Certain binary analysis techniques (including symbolic execution, which was used by almost every CGC competitor) essentially perform an emulation of binary code on an exotic domain (that is, instead of reasoning about ones and zeroes as a normal CPU would, they deal with symbolic expressions, value ranges, and so on). These techniques need to be provided with models for the functionality of the environment, to represent the side effects of the actions performed by system calls. Unfortunately, modern operating systems utilize a wide range of such system calls (Linux has more than 300, for example), which makes the creation of these models tedious.

DARPA worked around this problem by creating the DECREE operating system, a simplified OS that contains just seven system calls:

- **terminate:** the equivalent of Linux's `exit()`
- **transmit:** the equivalent of Linux's `send()`
- **receive:** the equivalent of Linux's `recv()`
- **fdwait:** the equivalent of Linux's `select()`
- **allocate:** the equivalent of Linux's `mmap()`
- **deallocate:** the equivalent of Linux's `munmap()`
- **random:** the equivalent of Linux's `get_random()`

By simplifying the environment model, DARPA greatly lowered the barrier to entry, removing much tedious engineering effort from the development of CRSs. Otherwise, the environment was standard, using the well-studied and well-supported x86 architecture and a simple, custom binary file format (supporting only statically linked binaries).

CGC Qualifying Event

Because the CGC attracted more than 100 prospective teams, DARPA held a qualifying round, dubbed the CGC Qualifying Event, or CQE. One of these prospective teams was Shellphish.

Shellphish is a disorganized collection of hackers at the University of California at Santa Barbara computer security lab, and while the CGC was tangentially related to some of our research at the time, we could not devote much time to it. Thus, our CGC effort was more or less on the back burner until we could no longer ignore it—about two and a half weeks before the qualifying event.

In those two and a half weeks, we built a fledgling CRS, laying the groundwork for ideas that later turned into Driller[2] and Ramblr.[3] We built a vulnerability detection engine that combined the fuzzing techniques pioneered by American Fuzzy Lop (AFL)[4] with the symbolic execution capabilities of the `angr` framework.[5] In addition, we developed a patching engine that supported both "general" patches (when the CRS couldn't find a specific vulnerability to patch) and "targeted" patches (when it could).

The CQE differed from the final event in several ways. First, humans were allowed to monitor, start, and restart the CRSs but were not allowed to gain and use any knowledge from the binaries themselves. This made it less necessary to have a "bulletproof" system, because we could respond to system crashes. Second, actual

exploitation was unnecessary in the CQE—triggering a crash counted as "exploiting" a binary. This made it easier on the teams, in that they did not have to write an auto-exploitation component until after the CQE, but it also meant that the teams' patches had to prevent binaries from crashing, rather than simply making crashes unexploitable. Third, each CRS operated in isolation—there were no "flags" to capture from opponents, and scoring was purely on the basis of the crashing of the reference binaries in the dataset and protection against the reference exploits.

The CQE comprised a set of roughly 130 previously unseen binaries that the various CRSs had to analyze without any human involvement. Our CRS was able to crash 42 and prevent crashes in 49 of the CQE binaries. This, combined with the relatively high performance of the patches (which impacted the score), was enough to qualify us for the final event, netting us $750,000 in prize money.

CGC Final Event

The CGC Final Event (CFE) was very different from the CQE. The CRSs faced one another, needing to craft actual exploits (not just crashes), generate advanced patches with little overhead, steal flags, and adapt to the opponents' actions. There was more than a year gap between the CQE and the CFE to give teams enough time to develop their systems. True to form (and, again, because of the realities of a research lab), we procrastinated until the last three months.

The CFE was an incredible spectacle, in which the seven finalist CRSs (housed in seven massive racks provided by DARPA) competed live on stage, in front of an audience of thousands of people and with live commentary by "sportscasters." The humans of the teams watched it from the "team area," a cluster of couches within sight of the stage, but separated by a government-certified air gap.

There was absolutely no human intervention. The CRSs had to start on their own, hack on their own, and adapt to problems on their own. It was a grueling day, analogous in some small way to having to wait outside an operating room, with absolutely no control over what happens behind closed doors.

In the end, the Mechanical Phish won third place, netting us another $750,000 in prize money.

Birthing a CRS

What motivated us was the challenge of producing a fully integrated and robust system based on the current state of the art in binary analysis research. The difficulty of this challenge comes from the deep divide between "state of the art" and "robust," not just in technical terms, but in subtle cultural terms, too. Research labs, hanging on to the state of the art, are not normally well-known for the production of robust software. Instead, the incentive structure tends to favor the rapid creation and evaluation of "research prototypes," which work just enough to evaluate a given concept before moving on to the next research goal. Competing in something as consuming as the CGC is not a typical activity for a research lab. As such, we faced organizational and human challenges far beyond what we had been prepared for. While these challenges are not the type of technical details generally found in a scientific magazine, they are an important reality on our road to cyber autonomy.

We had to tackle designing an incredibly robust infrastructure, on hardware that we would not be able to access for issue remediation, at "move-fast-and-break-things" speed. We had to build a system that worked, without human intervention, for 10 hours.

From Research Prototype to Reliable Software

As academics, we are always chasing beyond the current cutting edge, to explore the next frontier. Because of this, the mode of operation in academic research is often to rapidly achieve the minimally functional prototype of an idea (without concern for beautiful design or reliability), evaluate it on a meaningful dataset, and publish the result. Normally, labs do not have (and do not need) the software development practices, ubiquitous in the industry, that encourage the development of *good* code. In fact, the term "research-quality code" has come to refer to code that showcases an idea but is almost unusable outside a research experiment. GitHub is rife with this kind of academically produced code, leading to much suffering among industry developers and enthusiasts who try to adopt it.

However, plenty of labs go against the grain, and our long history of creating services and software that work (such as Wepawet,[6] Anubis,[7] and `angr`[5]) is an attempt to provide to the public at large *usable* research prototypes. In the context of the CGC, the problem was that we did not have sufficiently good software development practices. This had to change on the fly—over the course of the CGC, we adopted practices such as continuous integration, issue tracking, and even an attempt at code freezes.

While this process was difficult, it had a humongous eventual payoff for our research. The direct benefit from the CGC was an extreme improvement in the reliability and performance of our binary analysis framework, `angr`. Since then, these improvements have been put to work powering a plethora of other research projects, both from our lab and from labs and companies around the world.

Human Organization

Through a process reminiscent of natural selection, our team settled into several main roles. We had a strategic leader, who oversaw the long-term direction and did the "people managing" (that is, the professor); the tactical captain, who managed the daily technical direction; and four technical teams to handle the infrastructure, the base binary analysis framework (`angr`), exploitation, and patching. These teams were logical, rather than physical entities—many of our teammates worked on more than one team throughout the CGC. For example, overlap between the base analysis framework team and the patching or exploitation team was fairly common.

Our team had a dozen people, none of whom had ever built a CRS before, and as mentioned earlier, we compressed the creation of the CRS into just over three months. Thus, the Mechanical Phish consumed just about three person-years of development. This is an area where the companies that were participating in the CGC had an advantage—from our understanding, the corporate teams had fewer members but were able to dedicate the entire two years of the challenge to their CRS development. In the end, as always, time was the most precious resource.

Making Use of Non-Temporal Resources

Other than time, for which we could have designed a better usage distribution, we also had to properly utilize a number of other resources. For example, DARPA provided a cluster of 64 extremely powerful machines for the development, testing, and eventual deployment of our CRS. Upon receipt of this hardware, we had a very vague idea of how the final version of Mechanical Phish would work. Thus, we designed an extremely flexible infrastructure, in which resources were automatically allocated as needed, using modern cluster management software (specifically, Kubernetes[8]).

This introduced a number of challenges. First, Kubernetes was (and remains) under extremely active development, so the base of our CRS was a moving target and needed periodic rewrites. Second, some of the CGC tooling that DARPA released (to work with binaries for the DECREE platform) required kernel modifications and ran only in a 32-bit VM (rather than a 64-bit container), necessitating the development of quite a bit of magic (actually consuming several hacker-weeks of development) to run virtual machines from within Kubernetes pods.

Complications

Naturally, complications arose throughout the process. Some of these were caused by our own disorganization or our attempts to surf the bleeding edge of security. Others were uncertainty issues that likely arise with any new competition format. Of course, the autonomy requirement of the final event, and the resulting inability to fix even minor issues arising from potential unexpected events, greatly amplified the stress caused by these complications.

Closed infrastructure. The most important part of building a system that can function autonomously is testing. As the various CRSs would talk to a central service (dubbed the Team Infrastructure; TI) during the game, the availability of this TI was necessary to test our systems. However, to avoid specific attacks developed against the TI, DARPA did not provide it to us in a runnable form. Instead, it provided a separate, partial implementation, called the Virtual Competition. The Virtual Competition implemented the minimal set of capabilities to start a game but did not have any functionality to evaluate exploits, test patches, generate sample network traffic, or compute scores.

Teams had to implement their own extensions to the Virtual Competition to have a readily available testbed, and DARPA did provide a network specification to help with this. As a result, there was no guarantee that these extensions were correct, or that they functioned in the same way as the actual TI.

Sparring partner uncertainty. The Virtual Competition was not enough to thoroughly test our systems. DARPA's solution to this was a set of "sparring partner" sessions, during which the actual TI would become accessible.

There was only one such interface, and eight entities clamoring to use it—the seven competitors and the infrastructure team. To prevent data leaks between these entities, the sparring partner could only be accessible to one of them at a time and had to be wiped between sessions. The result was that the sparring partner would, at an unannounced time, become accessible for an average of 30 minutes before shutting down. Unless the CRS was up, working, and properly scanning for the TI, the sparring session would be missed (this happened depressingly frequently).

Sometimes, this would cause interesting situations. One sparring partner round started in the middle of a database migration, with the central database of our CRS offline. To avoid wasting the sparring session, we launched off components of the CRS by hand, coordinating between them with a *paper database*, shown in Figure 1.

Thirty minutes was enough for five game rounds—sufficient to test basic CRS functionality but not CRS reliability. This meant that reliability issues, requiring a functional TI to trigger over a large amount of rounds, were very hard to identify. Worse, these five rounds had

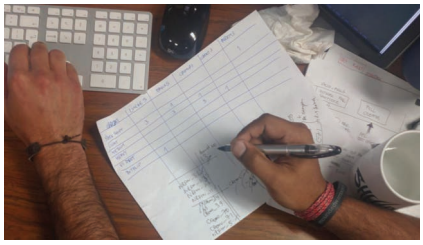

Figure 1. The paper database, used when a sparring partner session started in the middle of a database migration.

to be used to attempt to understand details of the performance scoring.

Performance scoring uncertainty. The CGC penalized teams for excessive performance overhead in their patched binaries. This performance score was critical—in fact, one of the teams that crashed the most binaries in the CQE failed to qualify precisely because their patches underperformed.

Naturally, determining the performance penalty was critical to the overall effectiveness of a CRS. However, two factors complicated this. First, DARPA did not specify exactly how performance overhead is calculated, and reproducing these calculations was very difficult. Furthermore, validating that the reproduced calculations were correct was impossible, as the relatively rare sparring partner sessions were the only way to get performance ground truth.

Second, DARPA kept the full penalty calculation formula (that took the time, memory, and file-size overhead and transformed it into a scaling factor applied to a team's points) secret. Without this formula, it was hard to reason about allowable performance tolerances for patching.

DARPA adopted this secrecy to stop teams from "gaming the system" ahead of time. This makes sense if there is a chance for human adaptability once the event begins. Without this chance, the secrecy made it extremely difficult for CRSs to make intelligent decisions about their patches.

Binary format uncertainty. One example of a small issue that caused great trepidation is the DECREE binary format itself. Into each CGC binary, the DECREE compiler tool-chain would insert a PDF file (which was always the same), along with a section of code that would checksum this PDF (by reading each byte of it when the program starts). This led to the obvious question: Will the PDF be in the binaries presented during the final event, as we remove the PDF to boost performance? Empirically, the answer to this question was "Yes," and the PDF seems to have been included to test whether the tools used were capable enough to properly remove it (and some binaries stressed the tools further by actually using data from the PDF outside of the checksum code) and reward such tools with lower performance overhead. However, DARPA stayed silent on the matter, requiring our tools to be adaptive to the no-PDF case, which did not manifest in the end.

Mechanical Phish

We have extensively described the various components of Mechanical Phish in research papers[2,3,5] and in an in-depth Phrack article.[1] For completion, we include a quick summary of the system high-level design. In Figure 2, one can see the overall layout of our architecture. The entire Mechanical Phish code base was composed by approximately 100,000 LoC (excluding external components), mostly written in Python. Out of these LoC, about 70,000 composed `angr`, the binary analysis framework on top of which most of the other components were built.

Infrastructure

DARPA provided every team 64 dedicated servers with overall:

- 1,280 physical cores,
- 16 TB of memory, and
- 64 TB of disk space.

To take full advantage of the hardware, we split our system into small, independent components and ran every component in a completely isolated environment. We used Docker containers to ensure components' isolation, ease of deployment, and scalability and Kubernetes to orchestrate the containers.

Every interaction with the game (for example, the retrieval of the current score or the submission of exploits) was performed through an API provided by DARPA, the TI. Ambassador and Network Dude were the components in charge of interacting with the game API and storing collected data into our central database, Farnsworth.

Meister and Scriba were the brain of our CRS. The first took care of reading the game status from Farnsworth and scheduling tasks. The second was responsible for deciding what patches and what exploits to submit, based on our internal evaluation and the feedback provided by the game API.

The hard program analysis work was done by the Workers, a set of components performing different tasks like bug finding, patching, exploitation, and results evaluation.

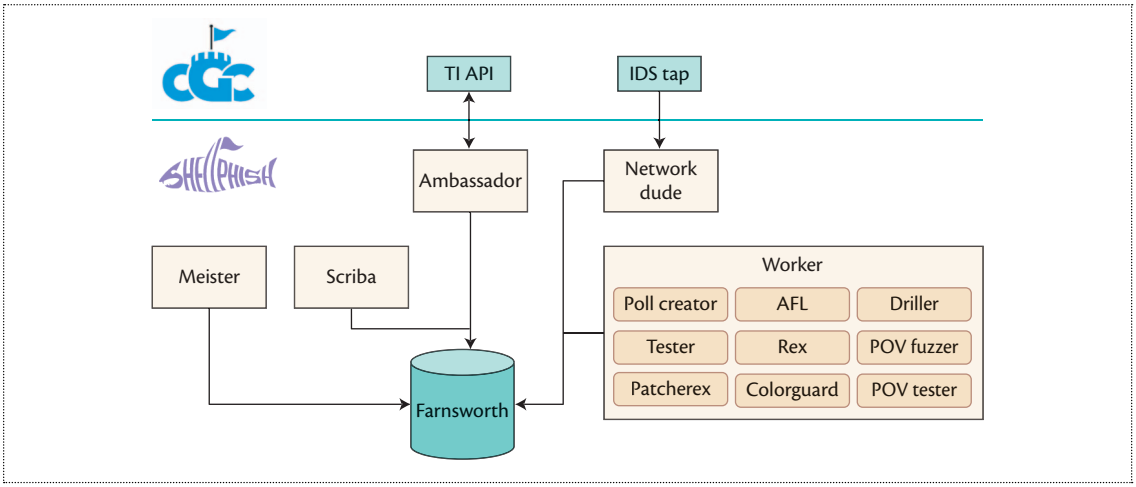

Figure 2. The architecture of Mechanical Phish.

Bug Finding

Mechanical Phish's exploitation involves two major steps. The first finds crashes in the target programs. The second step takes those crashes and attempts to figure out how they can be modified to produce exploits that take control of the program.

We used AFL, a well-known and highly successful evolutionary fuzzer, as the core of the bug-finding component of our CRS. For the CGC, we needed to handle a large variety of programs without any prior knowledge of what sort of inputs they will expect. An evolutionary fuzzer, such as AFL, is perfect for this, because it detects when inputs trigger new functionality inside the program, and then further mutates those inputs. This capability allows it to construct valid inputs, even when the program being fuzzed has strict requirements on the input format.

Although AFL is quite successful at finding bugs on its own, we found that it struggled to satisfy specific and difficult checks in the sample programs. Those checks could be as simple as matching a magic number or as difficult as solving an equation printed to the user. To handle these, we developed Driller, a tool that combines fuzzing with symbolic execution.[2] Symbolic execution is a slow but powerful technique for determining the equations that describe the state of the program at any point in execution. To use it efficiently, Driller limits the search space of the symbolic execution to that of the inputs generated by AFL. Specifically, the symbolic execution component will follow each input in AFL's corpus and check if there are any new locations in the program that it can reach.

AFL and symbolic execution, combined, made Driller highly successful in finding bugs that could be used to exploit the target programs.

Exploitation

The strategy we chose to exploit bugs found by the Driller component was to first analyze the crash using symbolic execution. That is, we symbolically traced the program following the crashing input, and when we got to the crash, we modified the input as needed to make an exploit.

In general, it can be extremely complicated to figure out how to exploit a particular bug or crash. For Mechanical Phish, instead of trying to design a general strategy, we came up with a list of crash types that we could exploit and methods to exploit just those particular crashes. The crashes we targeted were instruction pointer overwrite, arbitrary read address, arbitrary write address, and vtable overwrite. That list of crashes was picked specifically to target a fairly large range of what we expected to see in the CGC. Many types of bugs could map to the same crash type. For example, IP overwrite could occur from a buffer overflow, a use after free, an out-of-bounds index, and so forth. In addition, for each type of crash, there were multiple techniques that would try to exploit it in different ways.

After tracing the crash, Mechanical Phish would apply each technique and check if it succeeded in making a working exploit. Eventually, when one was found, Mechanical Phish would begin using it against the opponents.

Here we show a function with a basic stack overflow.

```
void say_hello() {
 char name[20];
 read_string(name);
 printf("hello %s\n", name);
 return;
}
```

In a normal interaction, the name provided will be short enough to fit entirely in the name buffer and

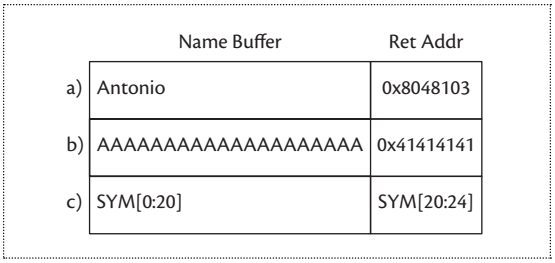

Figure 3. The function say_hello() from the listing in the main text has a buffer overflow. This figure shows the stack of the buffer during the following: (a) normal interaction, (b) overflowing input, and (c) the symbolically traced input.

the program will execute as expected. For that case, the stack of the function will look like the example in Figure 3a. However, the fuzzing component in Mechanical Phish can easily generate an input that is too long and overflows the return address, causing the program to crash (Figure 3b). Next, Mechanical Phish will symbolically trace the crashing input as shown in Figure 3c. It will understand that at the crash the instruction pointer is equal to `SYM[20:24]`, where `SYM` is used to denote symbolic input.

To exploit this, Mechanical Phish would try to jump to the bytes we control and execute them as code, referred to as *shellcode*. It added constraints to the equations that were collected during symbolic tracing. First, it placed the shellcode in memory by adding the constraint `SYM[0:20] == shellcode`. Then it constrained the overflowed return address to be the address of the shellcode, `SYM[20:24] == addr(shellcode)`. Finally, it asked the constraint solver to generate an input that matches these equations; this input was our exploit.

Patching

Patcherex, which is built on top of `angr`, is the central patching system of Mechanical Phish.

Patcherex follows an untargeted approach. In other words, it modifies binaries by applying generic binary hardening techniques, without using directly any knowledge about how a binary is exploitable. Nevertheless, in many cases, these hardening techniques are able to make vulnerabilities initially present not exploitable.

Furthermore, even when these vulnerabilities are still exploitable, the way in which exploits have to be carried out changes significantly in many patched binaries. For this reason, in many cases opponents were forced to analyze our patched binary to be able to adapt their exploits. However, we also implemented binary modifications hindering static and dynamic analysis of our patched binaries, making automatic analysis extremely hard, if not impossible. These included both passive countermeasures (that is, the produced binary files were slightly corrupted, being able to be executed in the DECREE environment but not analyzed with gdb or IDA) and active countermeasures. For example, we identified a buggy instruction in the floating-point support of the QEMU emulator that, when specific conditions were met, would cause the process to freeze. The inclusion of this instruction in our patches would hang any systems based on QEMU (in fact, the visualization system used by the organizers in the final event actually froze due to this countermeasure when visualizing an attempted exploit against one of our patched binaries).

Given the scoring system of the CGC competition, the primary concern while developing Patcherex was not to degrade the functionality of the original binaries and their performance. In fact, while it is reasonably easy, in general, to harden a binary to make it not exploitable, it is extremely hard to achieve this goal without significantly affecting its performance. Furthermore, compiled code often presents corner cases (due to, for instance, compiler optimizations) that, if not handled correctly during patching, will lead to the generation of nonfunctioning code.

Patcherex applies to any analyzed binary a list of techniques that corresponds to high-level patching strategies. Applying a specific technique to a binary generates a set of patches, which are low-level descriptions of how a fix or an improvement should be made on the target binary, such as adding some code or data to the original binary.

We implemented three different types of techniques:

- *Binary hardening*: Generic binary hardening techniques. For instance, we implemented encryption of the return pointer and a "loose" form of control flow integrity. We also implemented a technique to prevent memory-leaking exploits. In particular, we added code to the patched binary to check the transmitted data.
- *Anti-analysis*: Techniques aiming to prevent rivals from analyzing or stealing our patched binaries. For instance, we specifically added code triggering QEMU emulation bugs. In addition, we also inserted a back door in our patched binaries so that, in case they were reused by any opponent team, we could have trivially exploited them.
- *Binary optimization*: We realized that many of the provided binaries were easily optimizable (mainly because they were originally compiled without using compiler optimizations). Therefore, we applied binary optimization techniques (such as constant propagation or dead assignment elimination) to lower their memory/CPU usage. Improving the performance of the original binaries allowed us to lower the

negative impact (in terms of performance and, as a consequence, score) that the addition of patches generated by the previously mentioned techniques inevitably introduced.

Patches generated by applying the different techniques to a binary were then integrated into the original binary by a patching backend. Specifically, we developed a "reassembler" backend, able to convert a binary from its compiled form to an assembly form (recovering, for instance, function boundaries, function pointers, and pointers to data structures in memory). This form allows us to easily add or modify existing code and data and then use existing assemblers to generate a patched binary. Full details about the reassembler backend have been published in an academic paper.[3] As a fallback solution, in case the reassembler backed fails to generate a working patched binary, we use a different backend. This alternative approach is based on the inline insertion of detours (that is, *jmp* instructions), and it generates patched binaries that are less likely to misbehave, but slower and more memory greedy.

> **The drive to win against other teams is a stronger motivating force than a research deadline or the need to achieve some abstract result.**

Lessons Learned

Participating in the CGC taught us a number of lessons, both technical and nontechnical, which shaped our research and the pursuit of similar endeavors.

Teamwork

Effective teamwork is essential. A graduate student lab might not have the discipline of the well-managed development group of a company, but it has a unique drive and a camaraderie that cannot be easily replicated. Even though we suffered some setbacks due to the lack of experience in the development of high-quality software, the team was able to step up to the task without concerns about personal-life side effects. This is what a competition, like the CGC and many human-based CTFs, fosters: the drive to win against other teams is a stronger motivating force than a research deadline or the need to achieve some abstract result. On the other hand, these engagements cannot be the norm, as the toll (in terms of stress and pure physical exhaustion) that these kinds of events bring is not sustainable.

Gaming the Game

Understanding the nature and rules of the game is essential. Interestingly, the top-scoring system, Mayhem, had a dramatic failure in the middle of the competition, which prevented the system from finding new exploits against other teams. However, by not doing anything and simply passively defending, the Mayhem system was able to maintain its advantage against the other CRSs, winning the competition.

On our side, we were undecided between two different approaches to pushing patches. This was an important part of the game, as pushing a new version of a binary came with a one-round penalty in terms of defense points. As a result, pushing binaries too often could result in a substantial loss of points.

Our two possible approaches were the following:

- *Always-patch strategy*: Push patched binaries as soon as we were sure that their performance was acceptable.
- *Patch-if-exploited strategy*: Push patched binaries as soon as we were sure that they were performant enough and we had developed an exploit for the vulnerability.

The second approach was motivated by the fact that we assumed that most teams would have the same (or at least a very close) capability for exploitation. Under this assumption, the fact that we found an exploit for a specific target binary would imply that it was highly likely that other teams would have found an exploit as well, and therefore, it was reasonable to push a patched binary and take the associated defense penalty.

A few hours from the beginning of the competition, somewhat emboldened by the fact that our patching seemed to be highly effective with minimum performance overhead, we decided to push patched binaries as soon as we were able to produce them (that is, we chose to use the always-patch strategy). This decision resulted in a penalty that cost us the victory, as our postgame analysis revealed.

In this regard, it is very important to point out that every team could look back and consider things that they might have done differently. Understanding what the best strategy "would have been" is easy after the game is over. On the contrary, before the game, many aspects were unknown (for instance, how many challenges will be exploited), and therefore, choosing an optimal strategy was significantly harder.

Our postgame analysis was performed by computing scores for several simulated CGC rounds where the

Mechanical Phish undertook different strategies. The results are as follows:

- *Patch-if-exploited strategy*: We calculated our score with a patch strategy that would delay patches until after we launched exploits on the corresponding binary. In this case, our score would have been 271,506, putting us in first place.
- *Never-patch strategy*: We assumed that any time an exploit would be launched on a binary against any team, the exploit would be run against us during that round and all subsequent rounds. With this calculation, our score would have been 267,065, putting us in second place.
- *No-op strategy*: We ran an analysis similar to the never-patch strategy, but we also removed any exploitation-provided points. In this case, we would have scored 255,678 points, barely beating Shellphish and placing third in the CGC.

Exploitation

We were surprised that our exploitation system turned out to be the most effective of any competitor's during the final event, in terms of both the unique number of exploits produced and the number of times an exploit successfully worked. We exploited 15 different challenges, while the next best competitor exploited 11. However, there were 82 total programs, so what kept us from exploiting more? First, there were a good number of errors in our implementation. But, other than those, we believe that automatically exploiting bugs requires more than the "bag of techniques" approach we developed.

In exploitation, it is common that a human will carefully set up the program state, such that when the bug is triggered, structures and memory are already correctly set up. Our approaches did not have any way to backtrack and trigger the other functionality before that bug that would aid in setting up the state correctly. For some cases, this implies that we had to hope that the fuzzing component generated a crash where the state was already set up correctly, and this was not always the case.

Binary Patching

Many techniques exist for binary patching, including in-place bytes replacement, detouring, and so on, as well as systematic patching solutions like static binary rewriting techniques and dynamic binary instrumentation.[9–11] However, the CGC setting imposed some vital restrictions: The customized OS (DECREE), which has a very restricted set of system calls and a significant lack of system mechanisms (like process forking and debugging), made any dynamic approach unusable. Moreover, the tight overhead allowances for both performance and file size prevented us from applying many static binary rewriting techniques, which either unacceptably degrade the overall performance of patched binaries or add a noticeable amount of extra bytes to provide safety guarantees for rewriting. Hence, we picked *reassembling* (or *reassembleable disassembling*[12]) as the major binary rewriting technique and implemented Ramblr, with detouring as a fallback.

Some facts about the binaries in the CQE make reassembling a natural choice: All binaries are self-contained—no library is needed at all. Nearly all the binaries are compiled without any optimization flags switched on. Most of the binaries are relatively small compared to real-world targets, like word processors or browsers. Last but not least, only a few binaries are obfuscated, and it is not difficult to identify this problem and bail out. These facts made our CFG recovery and code data differentiation—which are the foundation of many static analyses, including reassembling—much easier.

After the CFE, we successfully applied Ramblr on more targets—including many CTF binaries—for binary rewriting and patching. Nevertheless, it is worth noting that all target binaries we rewrote using Ramblr were not considered "huge." We believe that using Ramblr on large or complex binaries will not yield a satisfactory result, as code data differentiation becomes harder when the code base gets larger, and our reassembling approach is best-effort and empirical—it does not provide safety guarantees (that is, it does not guarantee that no immediate value is treated as a pointer during reassembling). As we see it, providing safety guarantees is very difficult, if not entirely impossible. Therefore, Ramblr, in its current form, does not seem to be an ideal choice for rewriting large, real-world targets.

Infrastructure

Our bug-finding techniques pushed the limits of the bleeding-edge DARPA-provided servers. During our tests, processes died because the system ran out of memory regularly, and entire servers became unresponsive because of CPU-intensive workloads.

Normally, human intervention can mitigate these problems quite easily. However, during the CGC Final Event, our CRS had to run completely autonomously, which is why we invested a substantial amount of time in creating a highly available and fault-tolerant system.

Containers, which we orchestrated through Kubernetes, were the core foundation of the Mechanical Phish. To facilitate proper recovery without losing too much data, we designed our components to be stateless, and we broke down the complex functionality of our CRS into smaller components that executed separately, and whose results could be check-pointed and stored. Thanks to this design and by leveraging the tools that

Kubernetes provides, server failures were not critical. In fact, in our tests, Mechanical Phish kept exploiting and patching even if up to two-thirds of the cluster failed.

Unfortunately, stateless services are only one side of the coin, and the most important components are not stateless, namely the Kubernetes API server itself and our database. For these only two stateful components, we deployed multiple redundant instances with fail-over running on different nodes.

Interestingly, when we started to design the architecture of our CRS (in December 2015), Kubernetes was still in an early stage (version 1.0, July 2015). Since then, and most notably during our development process, Kubernetes has seen significant development and many improvements have been made. Although a blessing, this was a curse at the same time: constant API changes and updates broke compatibility, and our code base had to be dealt with on a regular basis.

Regardless of our problems during the development for Mechanical Phish, Kubernetes was easy to use and powerful. In fact, after our positive experience, we converted our research lab from a system where users have bare-metal servers allocated to them, to a container-based system where users request CPU and memory on an ephemeral basis, improving our overall resources utilization significantly and allowing research experiments of significantly larger scale than ever before.

The autonomous system determines which analysis actions need to be carried out by its components. Then, the system creates tasklets, some of which can be delegated to humans.

Aftermath: DEF CON CTF

When the DARPA CGC was announced, LegitBS,[13] who were the organizers of the 2016 DEF CON CTF, decided to structure the competition in a way that was identical to the CGC, so that the CRS that would win the CGC could compete against human teams. As a result, the Mayhem CRS was one of the teams playing in the 2016 DEF CON CTF.

However, Shellphish was the only team that qualified for both the CGC and the DEF CON CTF, and therefore, we had a unique opportunity: we could have Mechanical Phish play alongside humans.

Mechanical Phish was able to observe how humans (that is, the Shellphish team members) interacted with a target application when they were trying to find vulnerabilities. Then, the system used these interactions as seeds for its own vulnerability analysis process, with surprising results. On many occasions, the system was able to leverage the human inputs to reach "deep" into the application and identify vulnerabilities that could not have been identified without human assistance. Interestingly, the CRS did more than simply play backup to its human partners. Rather, it used human input to enhance its own ability and beat the humans to the punch: more than half the vulnerabilities found by the combined team were created by Mechanical Phish after leveraging human input to guide its analysis.

The successful interaction between the automated reasoning system and the human analysts prompted a key observation. Throughout the history of the field of vulnerability analysis, the principal paradigm has been the use of *tool-assisted human analysis*, in which human analysts would carry out the core analysis tasks, while utilizing automated techniques as an aid. In this case, the humans are the orchestrators of the analysis process, and they delegate specific tasks to specific tools, taking care of combining and composing the results of multiple tools. The CGC pushed a second approach: complete automation, where fully automated strategy routines utilized fully automated analyses to identify, exploit, and patch flaws in software. This inspired a third, heretofore unexplored model, which is *human-assisted automated analysis* of software. In this model, in an inverse of current techniques where most approaches see automated tools as an aid or extension to human analysts, human analysts can instead be used as an aid to automated vulnerability analysis systems.

Following this approach, the autonomous system determines which analysis actions need to be carried out by its components. Then, the system creates *tasklets*, some of which can be delegated to humans with different skill levels (for instance, experts or nonexperts). Even though this approach is still in its infancy, our preliminary results show that by orchestrating humans in a large-scale complex vulnerability analysis process, it is possible to identify vulnerabilities that would not be identified by purely automated means, shining a new light on one of the most challenging problems in program analysis.[14] ■

References

1. "Cyber Grand Shellphish," Shellphish, Jan. 2017; http://shellphish.net/cgc.
2. N. Stephens et al., "Driller: Augmenting Fuzzing through Selective Symbolic Execution," *Proceedings of the Network and Distributed System Security Symposium* (NDSS 16), 2016.

3. R. Wang et al., "Ramblr: Making Reassembly Great Again," *Proceedings of the Network and Distributed System Security Symposium* (NDSS 17), 2017.
4. M. Zalewski, "American Fuzzy Lop," 2017; http://lcamtuf.coredump.cx/afl.
5. Y. Shoshitaishvili et al., "(State of) The Art of War: Offensive Techniques in Binary Analysis," *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.
6. M. Cova, C. Kruegel, and G. Vigna, "Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code," *Proceedings of the 19th International Conference on World Wide Web* (WWW 10), 2010.
7. U. Bayer, C. Kruegel, and E. Kirda, *TTAnalyze: A Tool for Analyzing Malware*, EICAR, 2006; https://www.cs.ucsb.edu/~chris/research/doc/eicar06_ttanalyze.pdf.
8. "Kubernetes," Google, 2014; https://kubernetes.io.
9. C.-K. Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 05), 2005, vol. 40, p. 190.
10. N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 07), 2007, p. 89.
11. M. Smithson et al., "Static Binary Rewriting without Supplemental Information: Overcoming the Tradeoff between Coverage and Correctness," *Proceedings 20th Working Conference on Reverse Engineering* (WCRE 13), 2013, pp. 52–61.
12. S. Wang, P. Wang, and D. Wu, "Reassembleable Disassembling," *24th Usenix Security Symposium* (USENIX Security 15), 2015, pp. 627–642.
13. L.B. Syndicate, "Legitimate Business Syndicate CGC Documentation," 2015; https://cgc-docs.legitbs.net.
14. Y. Shoshitaishvili et al., "Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance," *Proceedings of the ACM Conference on Computer and Communication Security* (CCS 17), 2017.

Yan Shoshitaishvili is an assistant professor at Arizona State University, where he leads research into automated program analysis and vulnerability identification techniques. Contact at zardus@shellphish.net.

Antonio Bianchi is a PhD candidate at the University of California at Santa Barbara. Contact at antoniob@cs.ucsb.edu.

Kevin Borgolte is a PhD candidate in the Computer Science department at the University of California at Santa Barbara. Contact at cao@shellphish.net.

Amat Cama is a world-renowned hacker, having participated in countless CTFs around the globe. Contact at amatcama@gmail.com.

Jacopo Corbetta is a senior engineer at Qualcomm Product Security. At the time of this writing, he was an independent researcher. Contact at jacopo.corbetta@gmail.com.

Francesco Disperati is a senior software engineer at PayJunction. Contact at me@nebirhos.com

Audrey Dutcher is an undergraduate computer science researcher at the University of California at Santa Barbara. Contact at dutcher@cs.ucsb.edu.

John Grosen is a computer science major at the Massachusetts Institute of Technology. Contact at jmg@johngrosen.com.

Paul Grosen is a high school student, researcher in the Security group in the Department of Computer Science at the University of California at Santa Barbara, and a Shellphish member. Contact at pcgrosen@cs.ucsb.edu.

Aravind Machiry is a PhD candidate at the University of California at Santa Barbara. Contact at machiry@cs.ucsb.edu.

Chris Salls is a PhD student at the University of California at Santa Barbara, where he works on automated techniques to find memory corruption bugs. Contact at salls@cs.ucsb.edu.

Nick Stephens is a security researcher and a member of the Shellphish team. Contact at nick.d.stephens@gmail.com.

Ruoyu "Fish" Wang is a PhD candidate at University of California at Santa Barbara. Contact at fish@shellphish.net.

Giovanni Vigna is a professor in the Department of Computer Science at the University of California at Santa Barbara, the CTO at Lastline, Inc., and the founder of Shellphish. Contact at zanardi@shellphish.net.