.:: Cyber Grand Shellphish ::.

Papers:
saelo - Attacking JavaScript Engines: A case study of JavaScriptCore and CVE-2016-4622 (2016-10-27)
Team Shellphish - Cyber Grand Shellphish (2017-01-25)

Title : Cyber Grand Shellphish

Author : Team Shellphish

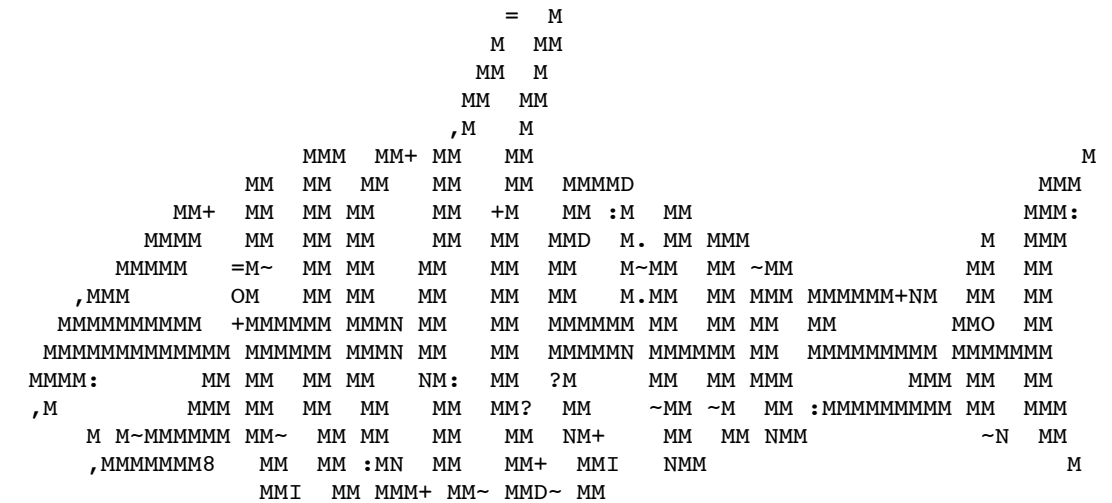
Date : January 25, 2017

```
|=-----------------------------------------------------------------------=|
|=---------------------=[ Cyber Grand Shellphish ]=----------------------=|
|=-----------------------------------------------------------------------=|
|=-------------------------=[ Team Shellphish ]=--------------------------=|
|=----------------------=[ team@shellphish.net ]=-------------------------=|
|=-----------------=[ http://shellphish.net/cgc#team ]=-------------------=|
|=-----------------------------------------------------------------------=|
```

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

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

```
                    MM
                  . MM
```



|=---=|
|=----------------------------=[The Team]=----------------------------=|
|=---=|
|=---=[zardus]=-----=[mike_pizza]=---=[anton00b]=----=[salls]=---=|
|=---=|
|=---=[fish]=---------=[nebhiros]=---=[cao]=--------=[donfos]=---=|
|=---=|
|=---=[hacopo]=---------=[nezorg]=---=[rhelmot]=------=[paul]=---=|
|=---=|
|=----------------------------=[zanardi]=----------------------------=|
|=---=|

Hacking is often considered more than a skill. In popular culture, hackers are seen as wizards of sorts, artists with powers to access the inaccessible, or perform acts that seem impossible. Hacking, like art, has great people, who are recognized for their skills, but whose abilities cannot be captured or reproduced. In fact, a single great hacker in a team is better than a hundred mediocre ones, similar to as none of the paintings from a hundred mediocre artists can match a painting from van Gogh.

Vulnerability analysis is the science of capturing and reproducing what some hackers do. Vulnerability analysis studies how one can reason, in a principled way, about finding and exploiting vulnerabilities in all types of software or hardware. By developing algorithms and tools to help humans identify flaws in software, researchers "codify" the knowledge that hackers use, in an organic way, to analyze systems and find their flaws. The resulting tools can then be used at scale, and composed to create new analysis systems.

This scientific process has generated a number of useful tools, such as static analysis tools, fuzzers, and symbolic execution frameworks. However, these tools codify only a subset of the skills of a hacker, and they are still used only to augment the abilities of humans.

One approach to push the codification of what human hackers do, is to take the hackers out of the equation. This is precisely what the DARPA Cyber Grand Challenge was set out to do.

The DARPA Cyber Grand Challenge (CGC) was designed as a Capture The Flag (CTF) competition among autonomous systems without any humans being involved. During the competition, Cyber Reasoning Systems (CRSs) would find vulnerabilities in binaries, exploit them, and generate patches to protect them from attacks, without any human involvement at all.

The separation between human and machine is key, as it forces the participants to codify, in algorithms, the techniques used for both attack and defense. Although the competition was only a first step toward capturing the art of hacking, it was an important one: for the first time, completely autonomous systems were hacking one another with code, and not human intuition, driving the discovery of flaws in complex software

systems.

Shellphish is a team that was founded by Professor Giovanni Vigna at UC Santa Barbara in 2005 to participate in the DEF CON CTF with his graduate students. Since then, Shellphish has evolved to include dozens of individuals (graduate students - now professors elsewhere, undergraduate students, visitors, their friends, etc.) who are somewhat connected by the Security Lab at UC Santa Barbara, but who are now spread all across the world. Nonetheless, Shellphish has never lost its "hackademic" background and its interest in the science behind hacking. Participation in many CTF competitions sparked novel research ideas, which, in addition to publications, resulted in tools, which, in turn, were put to good use during CTF competitions.

Given the academic focus of Shellphish, it is no surprise that the DARPA Cyber Grand Challenge seemed like a great opportunity to put the research carried out at the UC Santa Barbara SecLab to work. Unfortunately, when the call for participation for the funded track came out, the lab was (as usual) busy with a great number of research projects and research endeavors, and there were simply no cycles left to dedicate to this effort. However, when the call for the qualification round that was open to anybody who wanted to throw their hat in the ring was announced, a group of dedicated students from the SecLab decided to participate, entering the competition as team Shellphish.

With only a few weeks to spare, the Shellphish team put together a prototype of a system that automatically identifies crashes in binaries using a novel composition of fuzzing and symbolic execution. Unsurprisingly, the system was largely unstable and crashed more than the binaries it was supposed to crash. Yet, it performed well and Shellphish was one of the seven teams (out of more than a hundred participants) that qualified for the final event. Since Shellphish was not initially funded by DARPA through the funded track, it received a $750,000 award to fund the creation of the autonomous system that would participate in the final competition.

The following few months focused mostly on basic research, which resulted in a number of interesting scientific results in the field of binary analysis [Driller16, ArtOfWar16], and to the dramatic improvement of angr [angr], an open-source framework created at the UC Santa Barbara SecLab to support the analysis of binaries.

Eventually, the pressure to create a fully autonomous system increased to the point that academic research had to be traded for system-building. During the several months preceding the final competition event, all the energy of the Shellphish team focused on creating a solid system that could be resilient to failure, perform at scale, and be able not only to crash binaries, but also to generate reliable exploits and patches.

After gruesome months of Sushi-fueled work that lead to severe Circadian rhythm sleep disorder [Inversion] in many of the team's members, the Mechanical Phish Cyber Reasoning System was born. Mechanical Phish is a highly-available, distributed system that can identify flaws in DECREE binaries, generate exploits (called Proofs Of Vulnerability, or POVs), and patched binaries, without human intervention. In a way, Mechanical Phish represents a codification of some of the hacking skills of Shellphish.

Mechanical Phish participated in the final event, held in Las Vegas on August 4th, in conjunction with DEF CON, and placed third, winning a $750,000 prize. As a team, we were ecstatic: our system performed more successful exploits than any other CRS, and it was exploited on fewer challenges than any other CRS. Of course, in typical Shellphish style, the game strategy is where we lost points, but the technical aspects of our CRS were some of the best.

We decided to make our system completely open-source (at the time of writing, Shellphish is the only team that decided to do so), so that others can build upon and improve what we put together.

The rest of this article describes the design of our system, how it performed, and the many lessons learned in designing, implementing, and deploying Mechanical Phish.

--[Contents

--[001 - The Cyber Grand Challenge

The Cyber Grand Challenge was run by DARPA, and DARPA is a government agency. As such, the amount of rules, regulations, errata, and so forth was considerably out of the range with which we were familiar. For example, the Frequently Asked Questions document alone, which became the CGC Bible of sorts, reached 68 dense pages by the time the final event came around; roughly the size of a small novella. This novella held much crucial information, and this crucial information had to be absorbed by the team, digested, and regurgitated into the squawking maw of the Mechanical Phish.

--[001.001 - Game Format

The CGC Final Event took the form of a more-or-less traditional attack-defense CTF. This means that each team had to attack, and defend against, each other team. Counter to A&D CTF tradition, and similar to the setup of several editions of the UCSB iCTF, exploits could not be run directly against opponents, but had to be submitted to the organizers. Likewise, patches could not be directly installed (i.e., with something like scp), but had to be submitted through the central API, termed the "Team Interface" (TI) by DARPA. This allowed DARPA to maintain full control over when and how often attacks were launched, and how patches were evaluated.

The CGC was divided into rounds (specifically, there were 95 rounds in the final event) of at least 5 minutes each. Each round, the TI specified the currently-active challenges. A challenge could be introduced at any point, up to a maximum of 30 concurrently active challenges, and was guaranteed to remain live for a minimum of 10 rounds.

All teams would start with the same binaries for a challenge. Each round, teams could submit patches for their instances of the challenge binaries, exploits for the opponents' instances, and network rules to filter traffic. Submitted exploits would go "live" on the round *after* the one during which they were submitted, while submitted patches and network rules would go live 2 rounds after. Submitting a patch, causes that team to incur one round of downtime for that challenge, in which opponents can download the patched binary. Patched binaries were visible by opponents to allow them to exploit incomplete patches.

--[001.002 - Game Scoring

The CGC employed a scoring algorithm that had serious implications for the strategies that were viable for teams to adopt. Each team would be scored on a per-round basis, and a team's final score was the sum of all of their round scores. A team's round score was, in turn, the sum of their "Challenge Binary Round Scores" for that round. This CB Round Score was the real crux of the matter.

A CB Round Score was calculated for every Challenge Binary to be a simple multiplication of:

- Availability: this was a measure of the performance overhead and functionality impact introduced by a patch. The performance and functionality scores each ranged from 0 (broken) to 100 (perfect), and the availability score was the minimum of these numbers.
- Security: this number was 2 if the Challenge Binary was secure (i.e., it had not been exploited by any competitor that round), and 1 otherwise (if it was exploited by at least one competitor).
- Evaluation: this number ranged from 1.0 to 2.0, based on how many opponents the team was hitting with their exploits (if any) for this challenge binary. For example, if an exploit was succeeding against 2 of the 6 opponents, this number would be 1.33.

The round after a patch was submitted or an IDS rule was updated for a challenge binary, that challenge binary's availability score would be set to 0 and no exploits would be scheduled (either by the patching team or against the patching team) for that round.

On the surface, this seems like a simple formula. However, there were several complications:

- DARPA did not disclose the formula used to calculate the performance or functionality scores prior to the CFE. While this was presumably done to avoid letting teams "game the system", it led to an astonishing amount of uncertainty regarding patching. Teams knew that they had some "free" overhead (20% for the file size, and 5% each for runtime overhead and memory overhead), but did not know how additional overhead would be penalized.
- The runtime overhead included overhead introduced by the network IDS as it matched submitted network rules. This overhead was fundamentally unmeasurable before the game. During the CFE, it turned out that the overhead introduced by network rules from one service could actually influence overhead of *other* services. All this uncertainty kept all but two teams from using the network IDS during the CFE.
- The memory and runtime overhead was astonishingly hard to measure. As we discuss later, local measurements were so unreliable that we actually decided not to test our own patches for performance during the CFE, but optimized the techniques as much as possible beforehand and relied on the TI feedback for already-submitted patches. As we'll also discuss, DARPA themselves had some trouble keeping the measurements constant, with measurements between teams (for identical, unpatched binaries) varying by significant amounts.

Overall, the performance scoring was both a strength and weakness of the game. Because of the ease with which exploits can be "broken" by simple patches, it was critical to have some patch dis-incentives. However, the measurement of such patches is a very hard problem, and led to some chaos.

--[001.003 - Visibility

With the CGC, DARPA pioneered something called the "Consensus Evaluation". This had many implications for many parts of the game, but the biggest one was a slight change to the "visibility" of certain aspects of the game to the different teams. In this section, we'll detail what information was available to competitors during the game.

All interaction with the game was done via the CGC API (what DARPA called the Team Interface, or TI). The TI provided the following information:

- The Challenges: all teams could retrieve the challenges that were "in play" at any given moment.

- Patched Binaries: all teams could retrieve the patched binaries of *all* opponents. This prevented opponents from deploying "chump" patches that relied on simple security through obscurity.

- IDS rules: all teams could retrieve the IDS rules that were being fielded by their opponents.

- Availability impacts: a team could see the runtime overhead, memory

overhead, and functionality impact of their submitted patches.

- Crashes: a team could see what signals their binaries crashed with as a result of processing network traffic. This could give hints for whether or not exploits were being thrown against a challenge binary.

- Network Traffic: a team could get most of the network traffic, including all exploits and most of the functionality polls (DARPA reserved the right to keep some of them secret).

- POV Feedback: a team could get the results of their own exploits against other teams.

- Round scores: teams could get the overall round scores of all competitors.

This level of visibility is different than your "average" CTF. Specifically, the big difference is that the patches submitted by the Cyber Reasoning Systems were made available to opponents. In theory, this would allow opponents to analyze, and try to bypass, a CRS' patches. In practice, as we discussed in the Patching section, patched binaries were too "dangerous" to analyze. Teams were allowed to add anti-reversing code, and, in our experience, it was easy to find ways to crash analysis tools, or "clog" them, causing them to slow down or use extra resources. We did not dare to run our heavier analysis on the patched binaries, and we are not aware of any team that did. Instead, our heavy analysis was run strictly on the original, unpatched binaries, and the patched binaries were run through a quick analysis that would fix up memory offsets and so forth in the case of "chump" patches.

--[001.004 - Practice

Full autonomy is difficult to achieve. To mitigate some of this difficulty, DARPA provided CGC practice sessions through a "sparring partner". In the months leading up to the final event, the sparring partner would connect to our Cyber Reasoning System at random, unannounced times and begin a game.

The presence of the sparring partner was immensely helpful in debugging interactions with the API. However, the unannounced nature of these events meant that it was rarely possible to be ready for this debugging. Additionally, since DARPA only had one sparring partner setup, and teams had to take turns interacting with it, the sparring partner sessions were very short (generally, about 30 minutes, or 5 rounds). The reduced length of these sessions made it more difficult to truly stress-test the systems, and several teams showed signs of failure due to overload during the final event itself.

We mostly used the practice round to test the difference between DARPA's calculated overhead from our estimated one. This allowed us to get a vague idea of how DARPA calculated overhead, and tailor our patching techniques accordingly.

--[001.005 - DECREE OS

The binaries that made up the Cyber Grand Challenge's challenges were not standard Linux binaries. Instead, they were binaries for an OS, called DECREE, that was created specifically for the Cyber Grand Challenge. This OS was extremely simple: there were 7 system calls, and no persistence (i.e., filesystem storage, etc). The DECREE syscalls are:

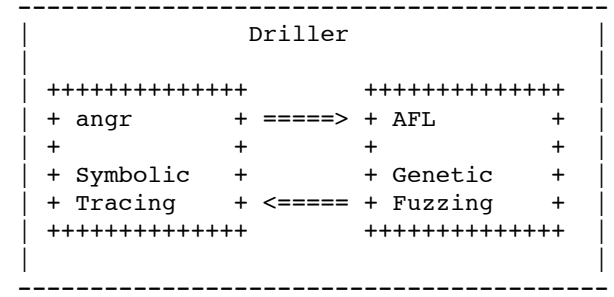
1. terminate: the equivalent of exit()
2. transmit: the equivalent of send()
3. receive: the equivalent of recv()
4. fdwait: the equivalent of select()
5. allocate: the equivalent of mmap()
6. deallocate: the equivalent of munmap()
7. random: a system call that would generate random data

The hardware platform for the binaries was 32-bit x86. Challenge authors were not allowed to include inline assembly code, only code which was produced by clang or included in a library provided by DARPA. The provided math library included instructions such as floating point logarithms and

trigonometric operations. All other code had to be produced directly by the compiler (clang).

This simple environment model allowed competitors to focus on their program analysis techniques, rather than the implementation details that go along with support complex OS environments. The DECREE OS was, in the opinion of many of us, the biggest thing that made the CGC possible with humanity's current level of technology.

--[002 - Finding Bugs



There are a few things we considered when we thought about how we wanted to find bugs in the Cyber Grand Challenge. Firstly, we will need to craft exploits using the bugs we find. As a result, we need to use techniques which generate inputs that trigger the bugs, not just point out that there could be a bug. Secondly, the bugs might be guarded by specific checks such as matching a command argument, password or checksum. Lastly, the programs which we need to analyze might be large, so we need techniques which scale well.

The automated bug finding techniques can be divided into three groups: static analysis, fuzzing, and symbolic execution. Static analysis is not too useful as it doesn't generate inputs which actually trigger the bug. Symbolic execution is great for generating inputs which pass difficult checks, however, it scales poorly in large programs. Fuzzing can handle fairly large programs, but struggles to get past difficult checks. The solution we came up with is to combine fuzzing and symbolic execution, into a state-of-the-art guided fuzzer, called Driller. Driller uses a mutational fuzzer to exercise components within the binary, and then uses symbolic execution to find inputs which can reach a different component.

* Fuzzing

Driller leverages a popular off-the-shelf fuzzer, American Fuzzy Lop. AFL uses instrumentation to identify the transitions that a particular input exercises when it is passed to the program. These transitions are tuples of source and destination basic blocks in the control flow graph. New transition tuples often represent functionality, or code paths, that has not been exercised before; logically, these inputs containing new transition tuples are prioritized by the fuzzer.

To facilitate the instrumentation, we use a fork of QEMU, which enables the execution of DECREE binaries. Some minor modifications were made to the fuzzer and to the emulation of DECREE binaries to enable faster fuzzing as well as finding deeper bugs:

- De-randomization
    Randomization by the program interferes with the fuzzer's evaluation of inputs - an input that hits an interesting transition with one random seed, may not hit it with a different random seed. Removing randomness allows the fuzzer to explore sections of the program which may be guarded by randomness such as "challenge-response" exchanges. During fuzzing, we ensure that the flag page is initialized with a constant seed, and the random system call always returns constant values so there is no randomness in the system. The Exploitation component of our CRS, is responsible for handling the removal of

randomness.

- Double Receive Failure
    The system call receive fails after the input has been completely read in by the program and the file descriptor is now closed. Any binary which does not check the error codes for this failure may enter an infinite loop, which slows down the fuzzer dramatically. To prevent this behavior, if the receive system call fails twice because the end-of-file has been reached, the program is terminated immediately.

- At-Receive Fork Server
    AFL employs a fork server, which forks the program for each execution of an input to speed up fuzzing by avoiding costly system calls and initialization. Given that the binary has been de-randomized as described above, all executions of it must be identical up until the first call to receive, which is the first point in which non-constant data may enter the system. This allows the fork-server to be moved from the entry of the program, to right before the first call to receive. If there is any costly initialization of globals and data structures, this modification speeds up the fuzzing process greatly.

* Network Seeds

Network traffic can contain valuable seeds which can be given to the fuzzer to greatly increase the fuzzing effectiveness. Functionality tests exercise deep functionality within the program and network traffic from exploits may exercise the particular functionality which is buggy. To generate seeds from the traffic, each input to the program is run with the instrumentation from QEMU to identify if it hits any transitions which have not been found before. If this condition is met, the input is considered interesting and is added as a seed for the fuzzer.

* Adding Symbolic Execution

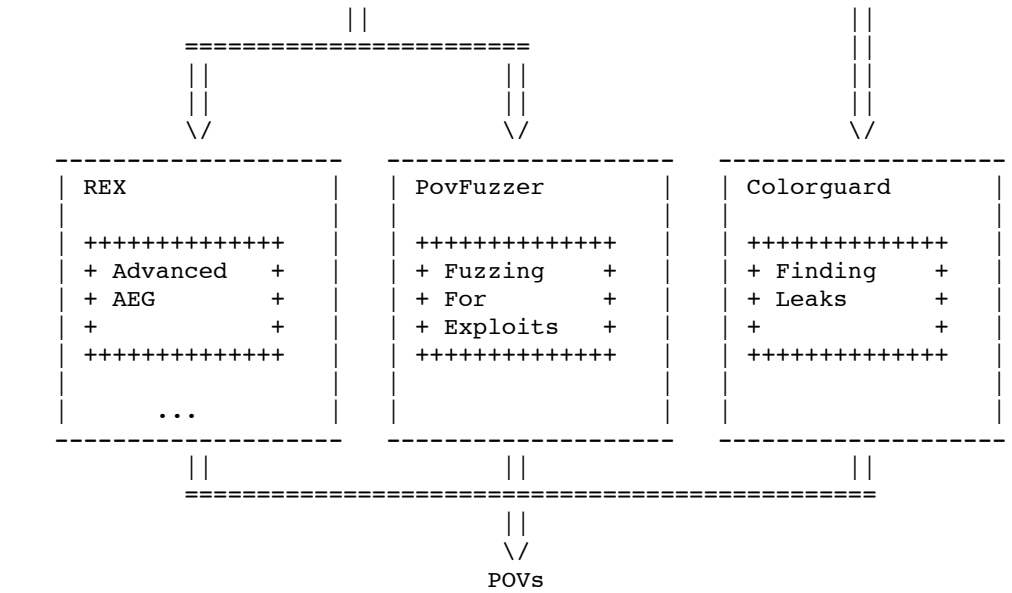
Although symbolic execution is slow and costly, it is extremely powerful. Symbolic execution uses a constraint solver to generate specific inputs which will exercise a given path in the binary. As such, it can produce the inputs which pass a difficult check such as a password, a magic number, or even a checksum. However, an approach based entirely on symbolic execution will quickly succumb to path explosion, as the number of paths through the binary exponentially increases with each branch.

Driller mitigates the path-explosion problem by only tracing the paths which the fuzzer, AFL, finds interesting. This set of paths is often small enough that tracing the inputs in it is feasible within the time constraints of the competition. During each symbolic trace, Driller attempts to identify transitions that have not yet been exercised by the fuzzer, and, if possible, it generates an input which will deviate from the trace and take a new transition instead. These new inputs are fed back into the fuzzer, where they will be further mutated to continue exercising deeper paths, following the new transitions.

* Symbolic Tracing

To ensure that the symbolic trace using angr's concolic execution engine is identical to the native execution trace, we use pre-constraining. In pre-constrained execution, each byte of input is constrained to match the original byte of input that was used by the fuzzer. When a branch is reached that would lead to a new transition, the pre-constraints are removed and then the solver is queried for an input which reaches the new transition. Pre-constraining also has the benefit of greatly improving execution time, because one does not need to perform expensive solves to determine the locations of reads and writes to memory because all variables have only one possible value.

--[003 - The Exploiting Component



* Exploitation

In the Cyber Grand Challenge, stealing flags is a little different than it is in ordinary Capture the Flag games. Instead of reading a secret "flag" file and submitting the contents to the organizers, an exploit is demonstrated by submitting a Proof of Vulnerability (POV), which the organizers will run.

A POV is a binary program, which will communicate with the opponent's binary and "exploit" it. There are two ways in which a POV can be considered successful:
- Type 1: Cause a segmentation fault where the instruction pointer and one additional register have values which were previously negotiated with the competition infrastructure. Must control at least 20 bits of EIP as well as 20 bits of another register.
- Type 2: Read 4 contiguous bytes from the secret flag data. The flag data is located at 0x4347c000-0x4347cfff and is randomly initialized by the kernel.

Different types of vulnerabilities might lend themselves to a specific type of POV. For example, a vulnerability where the user can control the address passed to puts() might only be usable as a Type 2 POV. On the other hand, a stack-based buffer overflow can clearly be used to create a Type 1 exploit simply by setting a register and EIP, but it can also be used to create a Type 2 exploit by using Return Oriented Programming or by jumping to shellcode which prints data from the flag page.

* Overview

The basic design for Mechanical Phish's automatic exploitation is to take crashes, triage them, and modify them to create exploits. Mechanical Phish does not need to understand the root cause of the bug, instead it only needs to identify what registers and memory it controls at crash time, and how those values can be set to produce a POV. We created two systems which are designed to go from crashes to exploits.

- PovFuzzer
    Executes the binary repeatedly, slightly modifying the input, tracking the relationship between input bytes and registers at the crash point. This method is fast, but cannot handle complex cases.

- Rex
    Symbolically executes the input, tracking formulas for all registers and memory values. Applies "techniques" on this state, such as jumping to shellcode, and return oriented programming to create a POV.

Now this design was missing one important thing. For some challenges the buggy functionality does not result in a crash! Consider a buffer over-read, where it reads passed the end of the buffer. This may not cause a crash, but if any copy of flag data is there, then it will print the secret data. To handle this, we added a third component.

- Colorguard
    Traces the execution of a binary with a particular input and
    checks for flag data being leaked out. If flag data is leaked, then
    it uses the symbolic formulas to determine if it can produce a valid
    POV.

--[003.001 - PovFuzzer

The PovFuzzer takes a crash and repeatedly changes a single byte at a time
until it can determine which bytes control the EIP as well as another
register. Then a Type 1 POV can be constructed which simply chooses the
input bytes that correspond to the negotiated EIP and register values,
inserts the bytes in the payload, and sends it to the target program.

For crashes which occur on a dereference of controlled data, the PovFuzzer
chooses bytes that cause the dereference to point to the flag page, in the
hope that flag data will be printed out. After pointing the dereference at
the flag page, it executes the program to check if flag data is printed
out. If so, it constructs a Type 2 POV using that input.

The PovFuzzer has many limitations. It cannot handle cases where register
values are computed in a non-trivial manner, such as through
multiplication. Furthermore it cannot handle construction of more complex
exploits such as jumping to shellcode, or where it needs to replay a random
value printed by the target program. Even so, it is useful for a couple
reasons. Firstly, it is much faster than Rex, because it only needs to
execute the program concretely. Secondly, although we don't like to admit
it, angr might still have occasional bugs, and the PovFuzzer is a good
fallback in those cases as it doesn't rely on angr.

--[003.002 - Rex

The general design of Rex is to take a crashing payload, use angr to
symbolically trace the program with the crashing payload, collecting
symbolic formulas for all memory and input along the way. Once we hit the
point where the program crashes, we stop tracing, but use the constraint
solver to pick values that either make the crash a valid POV, or avoid the
crash to explore further. There are many ways we can choose to constrain
the values at this point, each of which tries to exploit the program in a
different way. These methods of exploiting the program are called
"techniques".

One quick thing to note here is that we include a constraint solver in our
POVs. By including a constraint solver we can simply add all of the
constraints collected during tracing and exploitation into the POV and
then ask the constraint solver at runtime for a solution that matches the
negotiated values. The constraint solver, as well as angr, operates on
bit-vectors enabling the techniques to be bit-precise.

Here we will describe the various "techniques" which Rex employs.

- Circumstantial Exploit
    This technique is applicable for ip-overwrite crashes and is the
    simplest technique. It determines if at least 20 bits of the
    instruction pointer and one register are controlled by user input. If
    so, an exploit is constructed that will negotiate the register and ip
    values and then solve the constraints to determine the user input
    that sets them correctly.

- Shellcode Exploit
    Also only applicable to ip-overwrites, this technique will search for
    regions of executable memory that are controlled by user input. The
    largest region of controlled executable memory is chosen and the
    memory there is constrained to be a nop-sled followed by shellcode to
    either prove a type-1 or type-2 vulnerability. A shellcode exploit
    can function even if the user input only controls the ip value, and
    not an additional register.

- ROP Exploit
    Although the stack is executable by default, opponents might employ
    additional protections that prevent jumping to shellcode, such as

remapping the stack, primitive Address Randomization or even some form of Control Flow Integrity. Return Oriented Programming (ROP) can bypass incomplete defenses and still prove vulnerabilities for opponents that employ them. It is applicable for ip-overwrite crashes as long as there is user data near the stack pointer, or the binary contains a gadget to pivot the stack pointer to the user data.

- Arbitrary Read - Point to Flag
A crash that occurs when the program tries to dereference user-controlled data is considered an "arbitrary read". In some cases, by simply constraining the address that will be dereferenced to point at flag data, the flag data will be leaked to stdout, enabling the creation of a type-2 exploit. Point to Flag constrains the input to point at the flag page, or at any copy of flag data in memory, then uses Colorguard to determine if the new input causes an exploitable leak.

- Arbitrary Read/Write - Exploration
In some cases, the dereference of user-controlled data can lead to a more powerful exploit later. For example, a vtable overwrite will first appear as an arbitrary read, but if that memory address points to user data, then the read will result in a controlled ip. To explore arbitrary reads/writes for a better crash, the address of the read or write is constrained to point to user data, then the input is re-traced as a new crash.

- Write-What-Where
If the input from the user controls both the data being written and the address to which it is written, we want to identify valuable targets to overwrite. This is done by symbolically exploring the crash, to identify values in memory that influence the instruction pointer, such as return addresses or function pointers. Other valuable targets are pointers that are used to print data; overwriting these can lead to type-2 exploits.

--[003.003 - Colorguard

As explained above, there are some challenges which include vulnerabilities that can leak flag data, but do not cause a crash. One challenge here is that it is difficult to detect when a leak occurs. You can check if any 4 bytes of output data are contained in the flag page, but this will have false positives while fuzzing, and it will miss any case where the data is not leaked directly. The challenge authors seem to prefer to xor the data or otherwise obfuscate the leak, maybe to prevent such a method.

To accurately detect these leaks we chose to trace the inputs symbolically, using angr. However, symbolic tracing is far too slow to run on every input that the fuzzer generates. Instead, we only perform the symbolic tracing on the inputs which the fuzzer considers "interesting". The hope here is that the leak causing inputs have a new transition, or number of loops which is unique, and that the fuzzer will consider it interesting. There are definitely cases where this doesn't work, but it's a fairly good heuristic for reducing the number of traces.

In an effort to further combat the slowness of symbolic execution, Colorguard takes advantage of angr's concrete emulation mode. Since no modification of the input has to be made if a flag leak is discovered, our input is made entirely concrete, with the only symbolic data being that from the flag page. This allows us to only execute symbolically those basic blocks that touch the secret flag page contents.

Colorguard traces the entire input concretely and collects the symbolic expression for the data that is printed to stdout. The expression is parsed to identify any four consecutive bytes of the flag page that are contained in the output. For each set of bytes, the solver is queried to check if we can compute the values of the bytes only from the output data. If so, then an exploit is crafted which solves for these four bytes after receiving the output from the program execution.

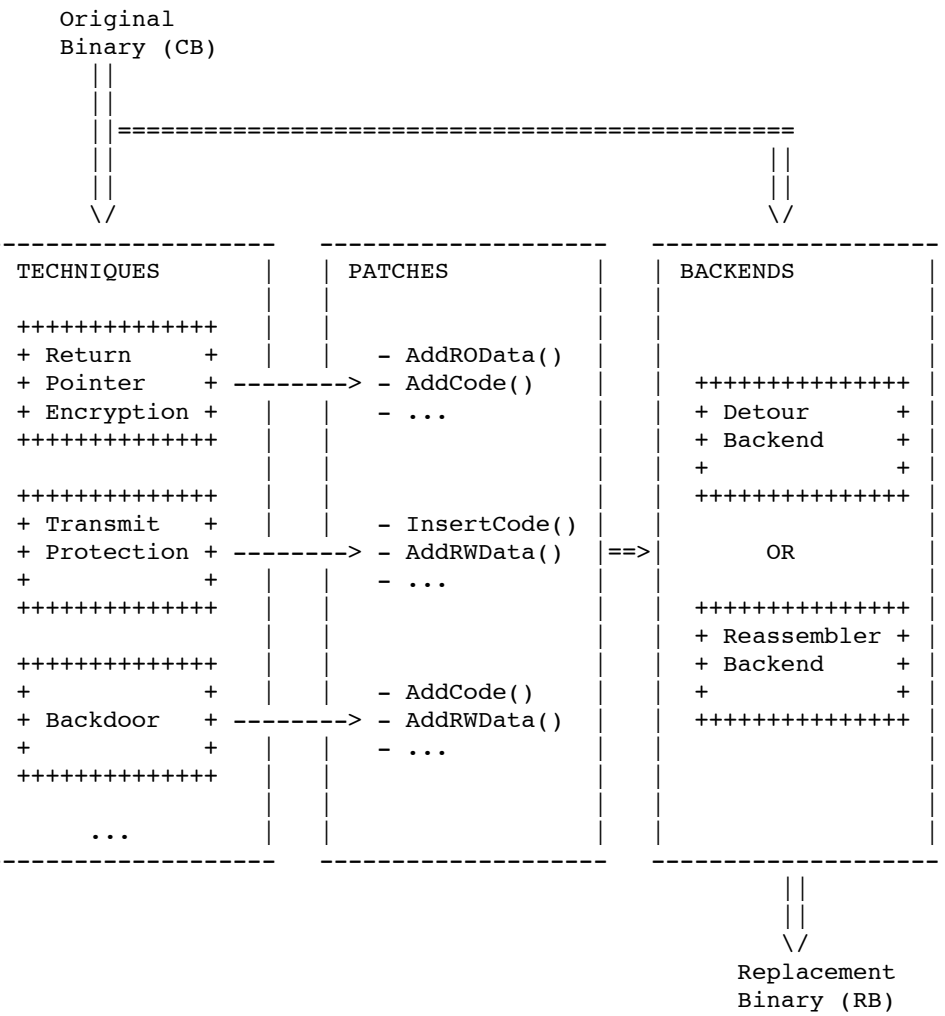
One caveat here is that challenges may use many bytes of the flag page as a random seed. In these cases we might see every byte of the flag page as part of the expression for stdout. Querying the constraint solver for

every one of these consecutive four byte sequences is prohibitively slow, so it is necessary to pre-filter such expressions. Colorguard does this pre-filter during the trace by replacing any expression containing more than 13 flag bytes with a new symbolic variable. The new symbolic variable is not considered a potential leak. The number 13 was arbitrarily chosen as it was high enough to still detect all of the leaks we had examples for, but low enough that checking for leaks was still fast.

--[003.004 - Challenge Response

A common pattern that still needs to be considered is where the binary randomly chooses a value, outputs it, and then requires the user to input that value or something computed from the random value. For example, the binary prints "Solve the equation: 6329*4291" and then the user must input "27157739". To handle these patterns in the exploit, we identify any constraints that involve both user input and random data. Once identified, we check the output that has been printed up to that point if it contains the random data. If so, then we have identified a challenge-response. We will include the output as a variable in the constraints that are passed to the exploit, and then read from stdout, adding constraints that the output bytes match what is received during the exploit. Then the solver can be queried to generate the input necessary for the correct "response".

--[004 - The Patching Component: Patcherex



Patcherex, which is built on top of angr, is the central patching system of Mechanical Phish. As illustrated in the overview image, Patcherex is composed of three major components: techniques, patches, and patching backends.

* Techniques

A technique is the implementation of a high-level patching strategy. A set

of patches (described below) with respect to a binary are generated after applying a technique on it. Currently Patcherex implements three different types of techniques:

- Generic binary hardening techniques, including Return Pointer Encryption, Transmit Protection, Simple Control-Flow Integrity, Indirect Control-Flow Integrity, Generic Pointer Encryption;
- Techniques aiming at preventing rivals from analyzing or stealing our patches, including backdoors, anti-analysis techniques, etc.;
- Optimization techniques that make binaries more performant, including constant propagation, dead assignment elimination, and redundant stack variables removal.

* Patches

A Patch is a low-level description of how a fix or an improvement should be made on the target binary. Patcherex defines a variety types of patches to perform tasks ranging from code/data insertion/removal to segment altering.

* Backends

A backend takes patches generated from one or more techniques, and applies them on the target binary. Two backends available in Patcherex:

- ReassemblerBackend: this backend takes a binary, completely disassembles the entire binary, symbolizes all code and data references among code and data regions, and then generate an assembly file. It then apply patches on the assembly file, and calls an external assembler (it was clang for CGC) to reassemble the patched assembly file to the final binary.
- DetourBackend: this backend acts as a fallback to the ReassemblerBackend. It performs in-line hooking and detouring to apply patches.

--[004.001 - Patching Techniques

In this section, we describe all techniques we implemented in Patcherex.

Obviously we tried to implement techniques that will prevent exploitation of the given CBs, by making their bugs not exploitable. In some cases, however, our techniques do not render the bugs completely unexploitable, but they still force an attacker to adapt its exploits to our RBs. For instance, some techniques introduce differences in the memory layout between our generated RB and the original CB an attacker may have used to develop its exploit. In addition, we try to prevent attackers from adapting exploits to our RB by adding anti-analysis techniques inside our generated binaries. Furthermore, although we put a significant effort in minimizing the speed and memory impact of our patches, it is often impossible to have performance impact lower than 5% (a CB score starts to be lowered when it has more than 5% of speed or memory overhead). For this reason we decided to "optimize" the produced RB, as we will explain later.

--[004.001.001 - Binary Hardening Techniques

We implemented some techniques for generic binary hardening. Those general hardening techniques, although not extremely complex, turned out to be very useful in the CFE.

Vulnerability-targeted hardening (targeted patching) was also planned initially. However, due to lack of manpower and fear for deploying replacement CBs too many times for the same challenge, we did not fully implement or test our targeted patching strategies.

* Return Pointer Encryption

This technique was designed to protect from classical stack buffer overflows, which typically give an attacker control over an overwritten saved return pointer. Our defense mechanism "encrypts" every return pointer saved on the stack when a function is called, by modifying the function's prologue. The encrypted pointer is then "decrypted" before every ret instruction terminating the same function. For encryption and decryption we simply xor'ed the saved return pointer with a nonce (randomly generated during program's startup). Since the added code is executed every time a function is called, we take special care in minimizing the performance

impact of this technique.

First of all, this technique is not applied in functions determined as safe. We classify a function as safe if one of these three conditions is true:

- The function does not access any stack buffer.
- The function is called by more than 5 different functions. In this case, we assume that the function is some standard "utility" function, and it is unlikely that it contains bugs. Even if it contains bugs, the performance cost of patching such a function is usually too high.
- The function is called by printf or free. Again, we assume that library functions are unlikely to contain bugs. These common library functions are identified by running the functions with test input and output pairs. This function identification functionality is offered by a separate component (the "Function identifier" mentioned in the "Warez" Section).

To further improve the performance, the code snippet that encrypts and decrypts the saved return pointer uses, when possible, a "free" register to perform its computations. This avoids saving and restoring the value of a register every time the injected code is executed. We identify free registers (at a specific code location) by looking for registers in which a write operation always happens before any read operation. This analysis is performed by exploring the binary's CFG in a depth-first manner, starting from the analyzed code location (i.e., the location where the code encrypting/decrypting the return pointer is injected).

Finally, to avoid negatively impacting the functionality of the binary, we did not patch functions in which the CFG reconstruction algorithm has problems in identifying the prologue and the epilogues. In fact, in those cases, it could happen that if an epilogue of the analyzed function is not identified, the encrypted return address will not be decrypted when that epilogue is reached, and consequently the program will use the still-encrypted return pointer on the stack as the return target. This scenario typically happens when the compiler applies tail-call optimizations by inserting jmp instructions at the end of a function.

* Transmit Protection

As a defense mechanism against Type 2 exploits, we inject code around the transmit syscall so that a binary is forbidden from transmitting any 4 contiguous bytes of the flag page. The injected code uses an array to keep track of the last transmitted bytes so that it can identify cases in which bytes of the flag page are leaked one at a time.

* Simple Control-Flow Integrity

To protect indirect control flow instructions (e.g., call eax, jmp ebx), we inject, before any instruction of this kind, code that checks specific properties of the target address (i.e., the address which the instruction pointer will be after the call or jump). The specific checks are:

- The target address must be an allocated address (to prevent an attacker from using the indirect control-flow instruction to directly perform a Type 1 attack). To do so, we try to read from the target address, so that if the address does not point to an allocated region the program will crash before the instruction pointer is modified. This prevents simple Type 1 exploits, because the attacker must control at least 20 bits of the instruction pointer, and it is unlikely (but not impossible) that the negotiated value will end up inside an allocated memory region.

- The target address must be inside the memory range where the binary's code is typically loaded (for simplicity, we consider a "potential code" address to be any below 0x4347c000). To avoid breaking programs that use dynamically allocated code we do not perform this check if we statically detect that the analyzed program calls the allocate syscall in a way which will create additional executable memory.

- The target address is not a pop instruction. As a partial mitigation against ROP attacks, we dynamically check if the target of indirect calls is a pop instruction, and terminate the program otherwise.

* Uninitialized Data Cleaning

For each function, we identify all of the instructions that read and write to stack variables, and the stack offset that is accessed. If there is any path through the CFG such that a stack variable is read before it is written, then we consider it possible that there is an uninitialized data usage. For each variable that is detected in an uninitialized data usage, we zero that variable by adding stack cleaning code at the beginning of the function.

* Stack Base Address Randomization

On program's startup we add a random value (which can assume any 16-byte aligned value between 16 and 1024) to the stack pointer address. This adds indeterminism to the position of the stack, hindering any exploit making assumptions on the program's stack layout.

* malloc Protection

To interfere with exploitation of heap overflows, if we are able to identify a malloc-like function inside the analyzed CB, we slightly modify its behavior. In particular, we change the amount of bytes allocated by a small, pseudo-random, value.

* printf Protection

For every printf-like function identified, such as printf, snprintf, etc., we ensure that the function is not used to perform a "format string" attack. Specifically, if the format string parameter is neither in the binary's read-only memory nor a string already present in the binary, we stop the execution of the binary if:
- The format string parameter contains a meta character (e.g., "%").
- The format string parameter points to the flag page.

--[004.001.002 - Adversarial Techniques

Certain techniques are introduced to prevent rivals from analyzing, or even running our RBs in a controlled environment, while leaving those RBs still able to run in the real game environment. These techniques are presented in this section.

* Anti-analysis

We add some code, executed before the original entry point of the binary, to interfere with analyses that other teams could perform on our patched binary. Specifically, we add code to:
- Detect if the binary is executed using QEMU or PIN. To do so, we probe the implementation of different aspects that are difficult to emulate correctly, such as segment registers, transmission of partially allocated memory regions, syscall error codes values in case of "double failure". In addition, we add some code triggering a previously unknown implementation bug in QEMU, making it stall. Specifically, during the development of our CRS, we found that QEMU, when using user-mode emulation, does not correctly handle taking the square root of an "un-normal" floating point number, that is, a nonzero 80-bit float whose explicit integer bit (the highest bit of the mantissa) is zero. When this happens, QEMU will hang forever. Because of this anti-QEMU patch, some of our RBs broke the live visualization during the CFE public event.
- Interfere with symbolic execution engines. Our adversarial code contains self-modifying code designed to be extremely hard to simulate correctly and efficiently by a symbolic execution engine. In addition, some added code is specifically designed to trigger "path explosion" conditions.
- Interfere with automatic exploitation systems. We add code to transmit the flag page to file descriptor 2 (stderr). Although data transmitted to this file descriptor is not sent to a CRS interacting with a binary, an opponent could mistakenly assume that any contiguous 4 bytes transmitted from the flag page constitutes a Type 2 exploit and thus fielding a POV trying to leverage this "fake" leak. In addition, we inject "fake backdoor" code. This code is triggered if a CRS sends a specific 4-byte sequence. When triggered, it reads from the CRS 8 bytes used to set the program instruction pointer and the value of a register. For this reason, this code looks like easily exploitable to create a Type 1 POV, however the read values are xor'ed with a random value, before being used to set the instruction pointer and the register, making this code non-exploitable.

In addition, to counteract an adversary fielding one of our own patched binaries as its own patched binary, we inject a backdoor in every fielded patched binary. This backdoor can be used by our CRS to exploit the patched binary we generate, but it is designed to be extremely hard to be exploited from other teams' CRSs. The backdoor is triggered when a specific 4-byte sequence is received. To detect this, the function wrapping the receive syscall is modified to keep track of the first 4 bytes a program receives. Once triggered, the backdoor sends to the CRS a "challenge" C (which is a 19-bit value), and the CRS responds with a response R (a 64-bit value). Then, the backdoor code checks if the following condition is true: first_32_bits_of(SHA1(pad(R,160))) == pad(C,32), where pad(A,N) is a function padding the input value A up to N bits by adding zeros.

The challenge can be easily solved by pre-computing all the possible responses, but it is impossible for an opponent's POV to compute a solution for the challenge within the game-imposed 10-second timeout.

--[004.001.003 - Optimization Techniques

Performance is a vital concern of our patching strategy. While we stress the necessity of optimizing all our patching techniques, some overhead cannot be avoided. From analyzing binaries collected from CQE and CFE samples, we noticed that most of them are compiled with O0, i.e., without optimization enabled. We do not know why organizers decided not to optimize most of the provided challenges, but we speculated that this may have been decided to leave room for optimizations and patching.

It is well-known that O0 and O1 binaries can have a huge difference in execution time. Fortunately, some of the optimization methods used in O1 are not that difficult to perform directly on binaries. Further, angr provides all necessary data-flow analysis techniques, which makes the whole optimization development easier. Finally, with the help of the ReassemblerBackend, we can easily fully remove instructions that we want to get rid of, without having to replace them with nops. Therefore, we implemented some basic in-line binary optimization techniques in order to optimize O0 binaries in CFE, which are described below.

- Constant Propagation. We propagate constants used as immediates in each instruction, and eliminate unnecessary mov instructions in assembly code.
- Dead Assignment Elimination. Many unnecessary assignments occur in unoptimized code. For example, in unoptimized code, when a function reads arguments passed from the stack, it will always make a copy of the argument into the local stack frame, without checking if the argument is modified or not in the local function. We perform a conservative check for cases where a parameter is not modified at all in a function and the copy-to-local-frame is unnecessary. In this case, the copy-to-local instruction is eliminated, and all references to the corresponding variable on the local stack frame are altered to reference the original parameter on the previous stack frame. Theoretically, we may break the locality, but we noticed some improvement in performance in our off-line tests.
- Redundant Stack Variable Removal. In unoptimized code, registers are not allocated optimally, and usually many registers end up not being used. We perform a data-flow analysis on individual functions, and try to replace stack variables with registers. This technique works well with variables accessed within tight loops. Empirically speaking, this technique contributes the most to the overall performance gain we have seen during testing.

Thanks to these optimizations, our patches often had *zero* overall performance overhead.

--[004.002 - Patches

The techniques presented in the previous section return, as an output, lists of patches. In Patcherex, a patch is a single modification to a binary.

The most important types of patches are:

- InsertCodePatch: add some code that is going to be executed before an

instruction at a specific address.
- AddEntryPointPatch: add some code that is going to be executed before the original entry point of the binary.
- AddCodePatch: add some code that other patches can use.
- AddRWData: add some readable and writable data that other patches can use.
- AddRODataPatch: add some read-only data that other patches can use.

Patches can refer to each other using an easy symbol system. For instance, code injected by an InsertCodePatch can contain an instruction like call check_function. In this example, this call instruction will call the code contained in an InsertCodePatch named check_function.

--[004.003 - Backends

We implemented two different backends to inject different patches. The DetourBackend adds patches by inserting jumps inside the original code, whereas the ReassemblerBacked adds code by disassembling and then reassembling the original binary. The DetourBackend generates bigger (thus using more memory) and slower binaries (and in some rare cases it cannot insert some patches), however it is slightly more reliable than the ReassemblerBackend (i.e., it breaks functionality in slightly less binaries).

* DetourBackend

This backend adds patches by inserting jumps inside the original code. To avoid breaking the original binary, information from the CFG of the binary is used to avoid placing the added jmp instruction in-between two basic blocks. The added jmp instruction points to an added code segment in which first the code overwritten by the added jmp and then the injected code is executed. At the end of the injected code, an additional jmp instruction brings the instruction pointer back to its normal flow.

In some cases, when the basic block that needs to be modified is too small, this backend may fail applying an InsertCodePatch. This requires special handling, since patches are not, in the general case, independent (a patch may require the presence of another patch not to break the functionality of a binary). For this reason, when this backend fails in inserting a patch, the patches "depending" from the failed one are not applied to the binary.

* ReassemblerBackend

ReassemblerBackend fully disassembles the target binary, applies all patches on the generated assembly, and then assembles the assembly back into a new binary. This is the primary patching backend we used in the CFE. Being able to fully reassemble binaries greatly reduces the performance hit introduced by our patches, and enables binary optimization, which improves the performance of our RBs even further. Also, reassembling usually changes base addresses and function offsets, which achieves a certain level of "security by obscurity" -- rivals will have to analyze our RBs if they want to properly adapt their code-reusing and data-reusing attacks.

We provide an empirical solution for binary reassembling that works on almost every binary from CQE and CFE samples. The technique is open-sourced as a component in angr, and, after the CGC competition, it has been extended to work with generic x86 and x86-64 Linux binaries.

A detailed explanation as well as evaluation of this technique is published as an academic paper [Ramblr17].

--[004.004 - Patching Strategy

We used Patcherex to generate, for every CB, three different Replacement Binaries (RBs):

- Detouring RB: this RB was generated by applying, using the DetourBackend, all the patches generated by the hardening and adversarial techniques presented previously.
- Reassembled RB: this RB was generated by applying the same patches used by the Detouring RB, but using the ReassemblerBackend instead of the DetourBackend.
- Optimized Reassembled RB: this RB was generated as the Reassembled one,

but, in addition all the patches generated by the optimization techniques were added.

These three patched RBs have been listed in order of decreasing performance overhead and decreasing reliability. In other words, the Detouring RB is the most reliable (i.e., it has the smallest probability of having broken functionality), but it has the highest performance overhead with respect to the original unpatched binary. On the contrary the Optimized Reassembled RB is the most likely to have broken functionality, but it has a lower performance impact.

--[004.005 - Replacement Binary Evaluation

* Pre-CFE Evaluation

During initial stages of Patcherex development, testing of the RBs was done by an in-house developed component called Tester. Internally, Tester uses cb-test, a utility provided by DARPA for testing a binary with pre-generated input and output pairs, called polls. We made modifications to cb-test, which enabled the testing of a binary and its associated IDS rules on a single machine, whereas the default cb-test needs 3-machines to test a binary with IDS rules.

Tester can perform both performance and functionality testing of the provided binary using the pre-generated polls for the corresponding binary using its Makefile. For functionality testing, given a binary, we randomly pick 10 polls and check that the binary passes all the polls. For performance testing, we compute the relative overhead of the provided binary against the unpatched one using all the polls. However, there was huge discrepancy (~10%) between the performance overhead computed by us and that provided during sparring partner sessions for the same binaries. Moreover, during the sparring partner sessions, we also noticed that performance numbers were different across different rounds for the same binaries. Because of these discrepancies and to be conservative, for every patching strategy, we computed the performance overhead as the maximum overhead across all rounds of sparring partner sessions during which RBs with corresponding patching strategy are fielded.

During internal testing we used all the available binaries publicly released on GitHub (some of which were designed for the CQE event, whereas others were sample CFE challenges). To further extend our test cases, we recompiled all the binaries using different compilation flags influencing the optimization level used by the compiler. In fact, we noticed that heavily optimized binaries (e.g., -O3), were significantly harder to analyze and to patch without breaking functionality. Specifically, we used the following compilations flags: -O0, -Os, -Oz, -O1, -O2, -O3, -Ofast. Interestingly, we noticed that some of the binaries, when recompiled with specific compilation flags, failed to work even when not patched.

During the final stages of Patcherex development, we noticed that almost all generated RBs never failed the functionality and that the performance overhead was reasonable except for a few binaries. This, combined with the discrepancy inherent in performance testing, led us not to use any in-depth testing of our replacement binaries during the CFE.

* CFE Evaluation

For every RB successfully generated, Patcherex first performs a quick test of their functionality. The test is designed to spot RBs that are broken by the patching strategy. In particular, Patcherex only checks that every generated RB does not crash when provided with a small test set of hardcoded input strings ("B", "\n", "\n\n\n\n\n\n", etc.)

We decided to perform only a minimal tests of the functionality because of for performance and reliability considerations.

--[004.006 - Qualification Round Approaches

It is worth mentioning that for the CGC qualification round, the rules were very different from the final round. Between this and the fact that our analysis tools were not yet mature it was necessary for us to approach patching very differently from the final round approaches previously described.

In the qualification round, the only criteria for "exploitation" was a crash. If you could crash a binary, it meant that you could exploit it, and if your binary could crash, it meant that you were vulnerable. Furthermore, the qualification scoring formula was such that your "defense" score, i.e. how many vulnerabilities your patch protected against, was a global multiplier for your score between zero and one. This meant that if you didn't submit a patch for a binary, or if your patch failed to protect against any of the vulnerabilities, you received zero points for that challenge, regardless of how well you were able to exploit it.

This is such an unconventional scoring system that when we analyzed the (publicly available) patches produced by other teams for the qualification round, we found that at least one qualifying team had a patching strategy such that whenever they discovered a crash, they patched the crashing instruction to simply call the exit syscall. This is technically not a crash, so the teams that did this did in fact receive defense points for the challenges, and accordingly did in fact receive a non-negligible score for effectively stubbing out any vaguely problematic part of the binary.

Our approaches were slightly more nuanced! There were two techniques we developed for the qualification round, one "general" technique, meaning that it could be applied to a program without any knowledge of the vulnerabilities in a binary, and one "targeted" technique, meaning that it was applied based on our CRS' knowledge of a vulnerability. Each of the techniques could produce several candidate patched binaries, so we had to choose which one to submit in the end - our choice was based on some rudimentary testing to try to ascertain if the binary could still crash, and if not, to assess the performance impact of the patch.

It is important to notice for CQE, the patched binaries were tested by the organizers against a fixed set of pre-generated exploits. For this reason, our patched binaries just had to prevent to be exploited when run against exploit developed for the original, unpatched, version of the program. In other words, the attacker had no way to adapt its exploits to our patches and so "security trough obscurity" techniques were extremely effective during the qualification event.

--[004.006.001 - Fidget

Our "general" patching technique for CQE was a tool called Fidget. Fidget was developed the summer prior to the announcement of the CGC for use in attack-defense CTFs. Its basic intuition is that the development of an attack makes a large number of very strong assumptions about the internal memory layout of a program, so in many cases simply tweaking the layout of stack variables is a reliable security-through-obscurity technique.

At the time of the CQE, Fidget was a tool capable of expanding function stack frames, putting unused space in between local variables stored on the stack. This is clearly not sufficient to prevent crashes, as is necessary for the strange qualification scoring formula. However, the tool had an additional mode that could control the amount of padding that was inserted; the mode that we used attempted to insert thousands of bytes of padding into a single stack frame! The idea here is, of course, that no overflow attack would ever include hundreds of bytes more than strictly necessary to cause a crash.

The primary issue with Fidget is that it's pretty hard to tell that accesses to different members or indexes of a variable are actually accesses to the same variable! It's pretty common for Fidget to patch a binary that uses local array and struct variables liberally, and the resulting patched binary is hilariously broken, crashing if you so much as blow on it. There are a huge number of heuristics we apply to try not to separate different accesses to the same variable, but in the end, variable detection and binary type inference are still open problems. As a result, Fidget also has a "safe mode" that does not try to pad the space in between variables, instead only padding the space between local variables and the saved base pointer and return address.

We originally planned to use Fidget in the final round, since it had the potential to disrupt exploits that overflow only from one local variable into an adjacent one, something that none of our final-round techniques can address. However, it was cut from our arsenal at the last minute upon the

discovery of a bug in Fidget that was more fundamental than our ability to fix it in the limited time available! Unfortunate...

--[004.006.002 - CGrex

If we know that it's possible for a binary to crash at a given instruction, why don't we just add some code to check if that specific instruction would try to access unmapped or otherwise unusable memory, and if so exit cleanly instead of crashing? This is exactly what CGrex does.

Our reassembler was not developed until several weeks before the CGC final event, so for the qualification round CGrex was implemented with a primitive version of what then became the DetourBackend. Once our CRS found a crash, the last good instruction pointer address was sent to CGrex, which produced a patched binary that replaced all crashing instructions with jumps to a special inserted section that used some quirks in some syscalls to determine if the given memory location was readable/writable/executable, and exit cleanly if the instruction would produce a crash.

More precisely, CGrex takes, as input, a list of POVs and a CB and it outputs a patched CB "immune" against the provided POVs. CGrex works in five steps:

1) Run the CGC binary against a given POV using a modified QEMU version
   with improved instruction trace logging and able to run CGC binaries.

2) Detect the instruction pointer where the POV generates a crash (the
   "culprit instruction").

3) Extract the symbolic expression of the memory accesses performed by the
   "culprit instruction" (by using Miasm). For instance, if the crashing
   instruction is mov eax, [ebx*4+2] the symbolic expression would be
   ebx*4+2.

4) Generate "checking" code that dynamically:
    - Compute the memory accesses that the "culprit instruction" is going
      to perform.
    - Verify that these memory accesses are within allocated memory regions
      (and so the "culprit instruction" is not going to crash). To
      understand if some memory is allocated or not CGrex "abuses" the
      return values of the random and fdwait syscalls.

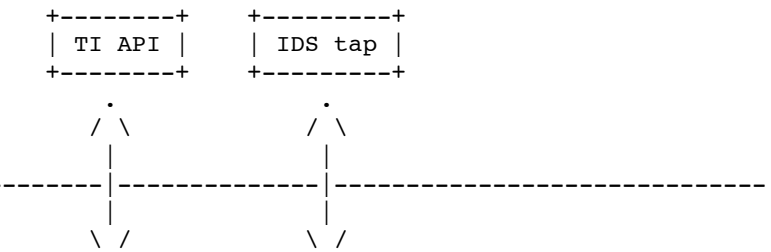
      In particular these syscalls were used by passing as one of the
      parameters the value to be checked. The kernel code handling these
      functions verifies that, for instance, the pointer were the number of
      random bytes returned by random is written is actually pointing to
      writable memory and it returns a specific error code if not. CGrex
      checks this error code to understand if the tested memory region is
      allocated. Special care is taken so that, no matter if the tested
      memory location is allocated or not, the injected syscall will not
      modify the state of the program.

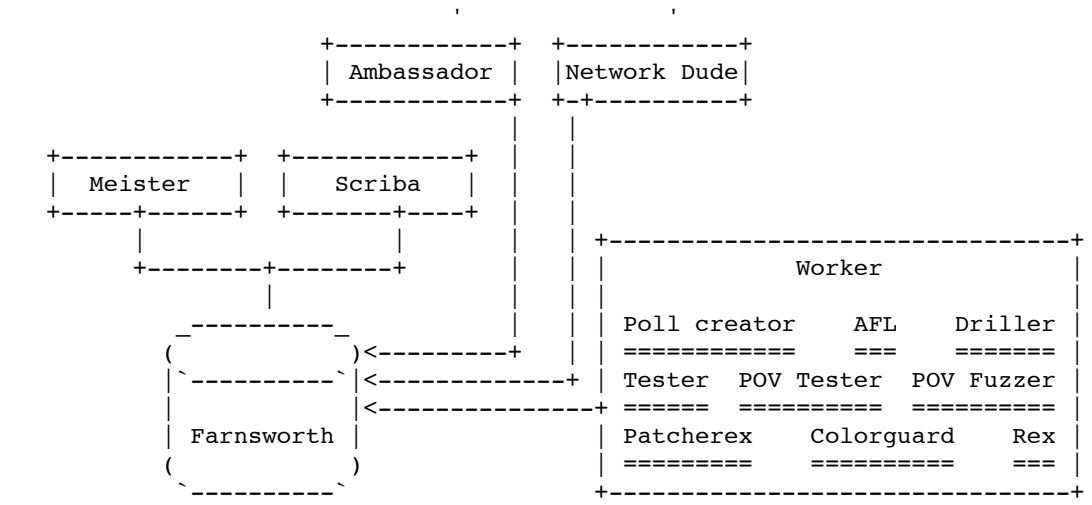
    - If a memory access outside allocated memory is detected, the injected
      code just calls exit.

5) Inject the "checking" code.

Steps 1 to 5 are repeated until the binary does not crash anymore with all the provided POVs.

--[005 - Orchestration





Designing a fully autonomous system is a challenging feat from an engineering perspective too. In the scope of the CFE, the CRS was required to run without fault for at least 10 hours. Although it was proposed to allow debugging during the CFE, eventually, no human intervention was permitted.

To that end, we designed our CRS using a microservice-based approach. Each logical part was split following the KISS principle ("Keep it simple, stupid") and the Unix philosophy ("Do one thing and do it well").

Specifically, the separation of logical units allowed us to test and work on every component in complete isolation. We leveraged Docker to run components independently, and Kubernetes to schedule, deploy, and control component instances across all 64 nodes provided to us.

--[005.001 - Components

Several different components of Mechanical Phish interacted closely together during the CRS (see diagram above). In the following, we will briefly talk about the role of each component.

* Farnsworth

  Farnsworth is a Python-based wrapper around the PostgreSQL database, and stores all data shared between components: CBs, POVs, crashing inputs, synchronization structures, etc. In our design, we prohibited any direct communication between components and required them to talk "through" Farnsworth. Therefore, Farnsworth was a potential single point of failure. To reduce the associated risk for the CFE, we paid particular attention to possible database problems and mitigated them accordingly.

* Ambassador

  Ambassador was the component that talked to the CGC Team Interface (TI) to retrieve CBs, obtain feedback, and submit RBs and POVs. In the spirit of KISS, this component is the only part of Mechanical Phish to communicate externally and the only source for the ground truth in respect to the game state.

* Meister

  Meister coordinated Mechanical Phish. For each component, a component-specific creator decided which jobs should be run at any point in the game, based on information obtained through Farnsworth and written by Ambassador. Consequently, Meister decided which jobs to run based on the priority information of each job (as specified by the creator) and usage of the nodes in terms of CPU and memory. Note that, specifically, Meister and its creators were entirely stateless. At any point, it could crash, yet it would not kill existing jobs upon automatic restart if they were still considered important by the creators.

* Scriba

An important task of the CRS was to select and submit the POVs and RBs, respectively. Scriba looked at performance results of exploits and patches and decided what and when to submit (for more details on the selection strategy see Section 6 - Strategy). As mentioned previously, after Scriba decided what and when to submit, Ambassador actually submitted to the TI (as Ambassador is the only component allowed to communicate externally).

* Network Dude

The Network Dude component received UDP traffic coming from the IDS tap and stored it into the database via Farnsworth. Since it is required to receive packets at line-speed, neither parsing nor analysis of the network data was performed within Network Dude, instead, we relied different components to process the network traffic.

* Worker

Worker was the executor for analysis tasks of the CRS. It wrapped tools such as angr, Driller, Patcherex in a generic interface to be managed easily. In fact, every Worker instance referred to an entry in a jobs queue specifying task arguments and type. Since some of the workers had to execute CGC DECREE binaries for functionality and performance evaluation, we included a DECREE virtual machine running on QEMU within a worker.

--[005.002 - Dynamic Resource Allocation

Another advantage of our architecture design, alongside dependency isolation and ease of deployment, was the possibility to dynamically scale components to meet our needs. Except for the PostgreSQL database and some internal Kubernetes services, all our components could run on any node without limitation.

Furthermore, when creating a job, Meister assigned it a priority based on the component and the current game status. For example, crash-generation jobs (Rex) were prioritized lower if an input crash was not considered reliable, or the analysis of a de-fielded CB was considered of no importance at all. Intuitively, all created jobs were sorted by descending values of priority, and scheduled through the Kubernetes API until all nodes' resources (CPU and memory) were saturated. Once all node resources were taken by running jobs, Meister killed jobs with lower priority to accommodate new higher priority jobs, but it did not over-provision.

--[005.003 - Fail-over

Mechanical Phish was required to run without failure for the duration of the CFE, an estimated 10 hours. Furthermore, no debugging sessions were permitted and the CRS was recommended to be resistant to minor hardware failure (or might have risked to "crash and burn").

To improve the reliability and resiliency of Mechanical Phish, we took various steps. First, every component, including Ambassador, Meister, Scriba, Network Dude, and Workers, was deployed as a Docker container. All components were designed to be entirely stateless, allowing us to restart them and move them across nodes if necessary.

Although components could be terminated abruptly without any significant consequence, some components were critical and were required to be running for Mechanical Phish to function correctly: These were Ambassador, Network Dude, Scriba, and Meister (crashing and recovering is acceptable for these components). Fortunately, Kubernetes provided a way to define always-running instances through DaemonSet and ReplicationController resources. If an instance of such type is terminated or timed out, it is automatically launched on another node (to prevent Kubernetes to be a single point-of-failure, Mechanical Phish was using a highly-available Kubernetes setup with multiple masters and virtual IP addresses for access).

* Database

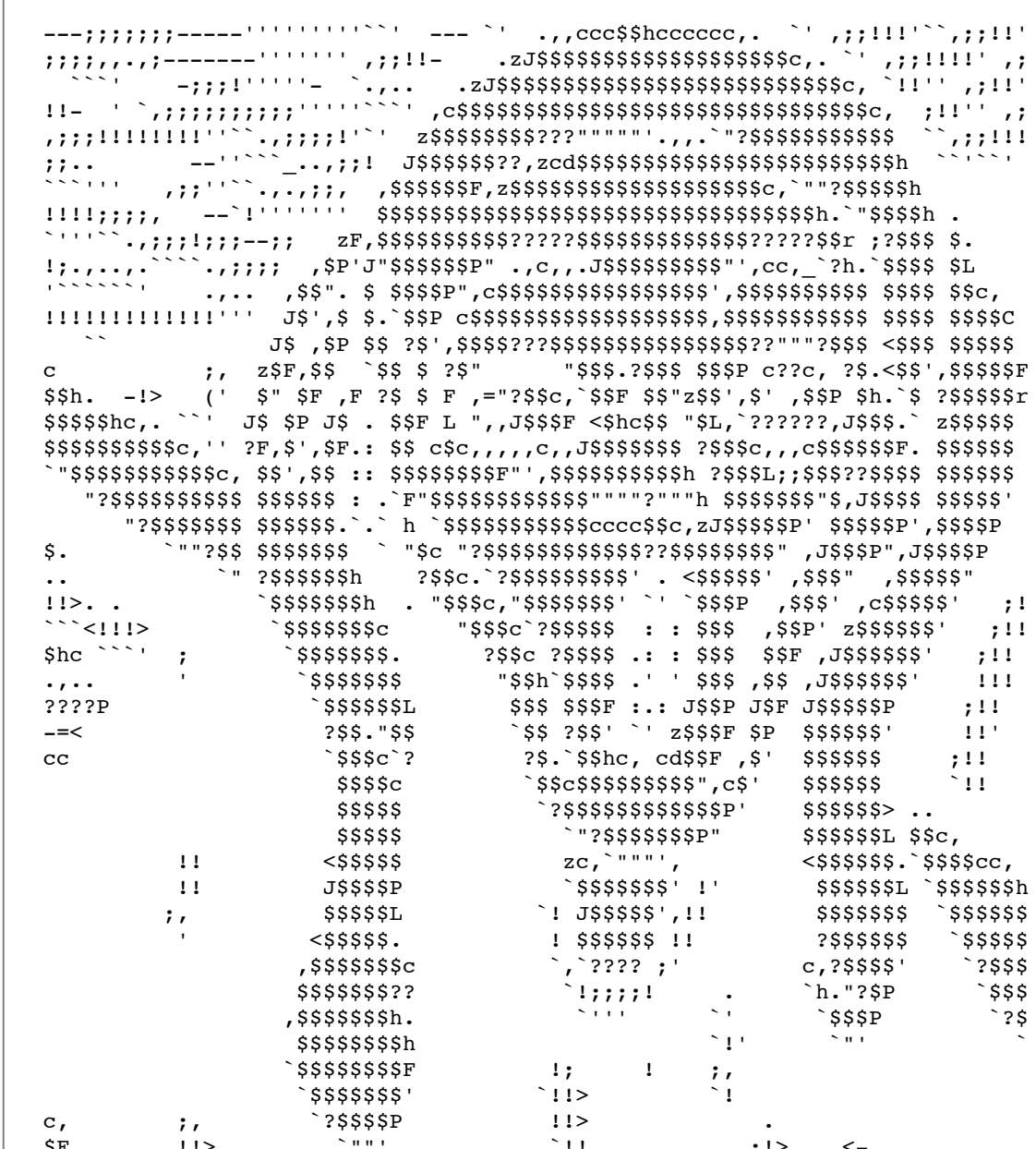
Naturally, the entire system cannot be completely stateless, and a single

stateful component is required, which was Farnsworth. To prevent any failure of the node running the PostgreSQL Docker containers or the containers themselves, we leveraged PostgreSQL's built-in master-slave streaming replication for a resilient system. Specifically, for the CFE, we ran 5 instances on 5 different physical nodes evenly spread across the rack, and an additional health-checking monitoring service. The monitor service itself was run using a ReplicationController resource. If the master database container would have been considered dead by the monitor, a slave instance would have been elected as the new master and a replacement slave would have been created on a healthy node. To prevent components from failing during database disaster recovery, they accessed the database in a retry loop with exponential back-off. In turn, it would have ensured that no data would have been lost during the transition from a slave to master.

* CGC Access Interfaces

The CGC CFE Trials Schedule defined that specific IP addresses were required to communicate with the CGC API. Given the distributed nature of our CRS and the recommendation to survive failure, the IP addresses remained the last single point of failure as specific components needed to be run on specific physical hosts. Consequently, we used Pacemaker and Corosync to monitor our components (Ambassador and Network Dude), and assign the specific IP addresses as virtual IP addresses to a healthy instance: if a node failed, the address would move to a healthy node.

--[006 - Strategy



The General Strategy of Shellphish

The Mechanical Phish was only about three months old when the final event of the Cyber Grand Challenge took place. Like any newborn, its strategic thought processes were not well developed and, unfortunately, the sleep-deprived hackers of Shellphish were haphazard teachers at best. In this section, we describe what amounted to our game strategy for the Mechanical Phish and how the rules of the Cyber Grand Challenge, combined with this strategy, impacted the final result.

* Development Strategy

The Shellphish CGC team was comprised completely of researchers at the UC Santa Barbara computer security lab. Unfortunately, research labs are extremely disorganized environments. Also unfortunately, as most of us were graduate students, and graduate students need to do research to survive (and eventually graduate), we were fairly limited in the amount of time that we could devote to the CGC. For example, for the CGC Qualification Event, we built our CRS in two and a half weeks. For the final event, we were able to devote a bit more time: on average, each member of the team (the size of which gradually increased from 10 to 13 over the course of the competition) probably spent just under three months on this insanity.

One thing that bit us is that we put off true integration of all of the components until the last minute. This led to many last-minute performance issues, some of which we did not sort out before the CFE.

* Exploitation Strategy

Our exploitation strategy was simple: we attack as soon and as frequently as possible. The only reason that we found to hold back was to avoid letting a victim team steal an exploit. However, there was not enough information in the consensus evaluation to determine whether a team did or did not have an exploit for a given service (and attempting to recover this fact from the network traffic was unreliable), so we decided on a "Total War" approach.

* Patching Strategy

For every not-failing RB, we submitted the patch considered as the most likely to have a better performance score. Specifically, given the way in which RBs were created, we ranked RBs according to the following list: Optimized Reassembled RB, Reassembled RB, Detouring RB. This choice was motivated by the fact that detouring is slow (as it causes cache flushes due to its propensity to jumping to many code locations), whereas the generated optimized RBs are fast. Of course, we could not always rely on the reassembler (and optimizer) to produce a patch, as these backends had some failure cases.

For every patch submitted, the corresponding CS is marked down for a round. Because this can be (and, in the end, was) debilitating to our score, we evaluated many strategies with regards to patching during the months before the CFE. We identified four potential strategies:

- Never patch: The simplest strategy was to never patch. This has the advantage of nullifying the chances of functionality breakages and avoiding the downtime associated with patching.

- Patch when exploited: The optimal strategy would be to patch as soon as we detect that a CS was being exploited. Unfortunately, detecting when a CS is exploited is very difficult. For example, while the consensus evaluation does provide signals that the binaries cause, these signals do not necessarily correlate to exploitation. Furthermore, replaying incoming traffic to detect exploitation is non-trivial due to complex behaviors on the part of the challenge binaries.

- Patch after attacking: An alternative strategy is to assume that an opponent can quickly steal our exploits, and submit a patch immediately after such exploits are fired. In our latter analysis, we determined that this would have been the optimal strategy, granting us first place.

- Always patch: If working under the assumption that the majority of
 challenge sets are exploited, it makes sense to always patch.

Most teams in the Cyber Grand Challenge took this decision very seriously. Of course, being the rag-tag group of hackers that we are, we did some fast-and-loose calculations and made a result based on data that turned out to be incorrect. Specifically, we assumed that a similar fraction of the challenges would be exploited as the fraction of challenges crashed during the CQE (about 70%). At the time, this matched up with the percentage of sample CGC binaries, provided by DARPA during sparring partner rounds, that we were exploiting. Running the numbers under this assumption led us to adopt the "always patch" approach:

1. At the second round of a challenge set's existence, we would check if we
  had an exploit ready. If not, we would patch immediately, with the best
  available patch (out of the three discussed above). Otherwise, we would
  delay for a round so that our exploit had a chance to be run against
  other teams.

2. Once a patch was deployed, we would monitor performance feedback.

3. If feedback slipped below a set threshold, we would revert to the
  original binary and never patch again.

In this way, we would patch *once* for every binary. As we discuss later, this turned out to be one of the worst strategies that we could have taken.

--[007 - Fruits of our Labors

In early August, our creation had to fight for its life, on stage, in front of thousands of people. The Mechanical Phish fought well and won third place, netting us $750,000 dollars and cementing our unexpected place as the richest CTF team in the world (with a combined winnings of 1.5 million dollars, Shellphish is the first millionaire CTF team in history!).

This is really cool, but it isn't the whole story. The CGC generated enormous amounts of data, and to truly understand what happened in the final event, we need to delve into it. In this section, we'll talk specifically regarding what happened, who achieved what, and how the CGC Final Event played out.

For reference throughout this section, the final scores of the CGC Final Event were:

Team	CRS Name	Points
ForAllSecure	Mayhem	270,042
TECHx	Xandra	262,036
Shellphish	Mechanical Phish	254,452
DeepRed	Rubeus	251,759
CodeJitsu	Galactica	247,534
CSDS	Jima	246,437
Disekt	Crspy	236,248

The attentive reader will notice that the score of the Mechanical Phish is the only one that is a palindrome.

--[007.001 - Bugs

The CGC was the first time that autonomous systems faced each other in a no-humans-allowed competition. As such, all of the Cyber Reasoning Systems likely faced some amount of bugs during the CFE. The most visible was Mayhem's, which resulted in the system being off-line for most of the second half of the game (although, as we discuss later in this section, that might not have hurt the system as much as one would think). Our system was no different. In looking through the results, we identified a number of bugs that the Mechanical Phish ran into during the CFE:

* Multi-CB pipeline assertions

We used fuzzing to identify crashes and POV Fuzzing to fuzz those crashes into POVs for challenge sets comprising of multiple binaries. Unfortunately, an accidental assert statement placed in the POV Fuzzing code caused it to opt out of any tasks involving multi-CB challenge sets, disabling our multi-CB exploitation capability.

* Network traffic synchronization

Due to a bug, the component that scheduled tasks to synchronize and analyze network traffic was set to download *all* recorded network traffic every minute. The volume of this traffic quickly caused it to exceed scheduling timeouts, and Mechanical Phish only analyzed network traffic for the first 15 rounds of the game.

* RB submission race condition

Due to a race condition between the component that determined what patches to submit and the component that actually submitted them, we had several instances where we submitted different patched binaries across different rounds, causing multiple rounds of lost uptime.

* Scheduling issues

Throughout the CFE, Mechanical Phish identified exploitable crashes in over 40 binaries. However, only 15 exploits were generated. Part, but not all, of this was due to the multi-CB pipeline assertions. It seems that the rest was due to scheduling issues that we have not yet been able to identify.

* Slow task spawning

Our configuration of Kubernetes was unable to spawn tasks quickly enough to keep up with the job queue. Luckily, we identified this bug a few days before the CFE and put in some workarounds, though we did not have time to fix the root cause. This bug caused us to under-utilize our infrastructure.

--[007.002 - Pwning Kings

Over the course of the CGC Final Event, the Mechanical Phish pwned the most challenges out of the competitors and stole the most flags. This was incredible to see, and is an achievement that we are extremely proud of. Furthermore, the Mechanical Phish stole the most flags even when taking into account only the first 49 rounds, to allow for the fact that Mayhem submitted its last flag on round 49.

We've collected the results in a helpful chart, with the teams sorted by the total amount of flags that the teams captured throughout the game.

Team	Flags Captured (first 49/all)	CSes Pwned (first 49/all)
Shellphish	206 / 402	6 / 15
CodeJitsu	59 / 392	3 / 9
DeepRed	154 / 265	3 / 6
TECHx	66 / 214	2 / 4
Disekt	101 / 210	5 / 6
ForAllSecure	185 / 187	10 / 11
CSDS	20 / 22	1 / 2

Interestingly, Mayhem exploited an enormous amount of binaries before it went down, but the Mechanical Phish still achieved a higher exploitation score in the rounds that Mayhem was alive. One possibility is that the exploits launched by the Mechanical Phish were more reliable than those of Mayhem. However, its raw exploitation power should not be underestimated: within 49 rounds, before going off the grid, Mayhem managed to exploit 10 binaries. While the Mechanical Phish surpassed it over the entire game, this is still quite impressive.

--[007.003 - Patching Kings

To understand how effective our patches were, we calculated the number of flags lost (and the number of CSes on which flags were lost) throughout the game, both for the first 49 rounds (in which Mayhem was online) and for the entire game. As expected, because the Mechanical Phish patched every binary, we found that it was the least-exploited CRS. Specifically, it only leaked flags on 12 challenge sets. The runner-up in this sense (the second-place team, TECHx), leaked flags on 14 binaries.

Interestingly, TECHx beat us on the total number of flags leaked. This makes sense -- they likely had some analysis to identify *when* they were being exploited, and patch only then. And, likely, their patches were more effective than our own, leading to a lower number of total flags leaked!

The results, sorted by the number of compromised challenges, are:

Team	Defended CS-Rounds (49/all)	CSes Compromised (49/*)
Shellphish	29 / 68	7 / 12
TECHx	27 / 61	7 / 14
DeepRed	32 / 87	6 / 15
ForAllSecure	54 / 160	7 / 16
CodeJitsu	61 / 104	9 / 16
Disekt	66 / 127	9 / 17
CSDS	108 / 189	9 / 18

--[007.004 - Patching Effectiveness

Before the CFE, we tested our patches against the dataset of all CQE binaries released until that point, compiled with many different compilation flags. On this dataset, our chance of breaking a binary, or lowering performance to below 85% (the guesstimated threshold at which we would pull the binary back) was under 2%, and we decided to abandon patch performance/functionality testing. On the actual CFE dataset, we fared worse.

The CFE had 82 challenge sets. For these 82 challenge sets, we successfully generated patches for 78. Of these 78, we had to pull back 10 because the patch broke the binary and 7 because the patch lowered the performance of the binary below acceptable levels. This was considerably higher than expected. Furthermore, due to a race condition between the patch submission logic and the component that retrieves the game state, we did end up with situations where we would submit *multiple* patches, thinking that the round had not ended whereas, in reality, it had. In total, we made 107 RCB submissions throughout the game. This was the second-highest amount, with only CodeJitsu submitting more (108).

Because very few challenges were exploited during the CFE, this was unequivocally the wrong choice. In total, we lost around 17000 points during the "downtime" rounds after the patch submissions alone.

--[007.005 - Alternate History Case Studies

Hindsight is 20/20. Even though we know, now, that we made the wrong choice regarding our patching strategy, it's still interesting to see what "could have been". In this section, we do that. To better understand the impact of our strategy decisions, we compute scores for several simulated CGC rounds where the Mechanical Phish undertook different strategies, and see what would have happened.

It's very important to point out that this is all fantasy. *Every* team can look back and consider things that they might have done differently. The most obvious one is Mayhem: had they avoided crashing, they might have absolutely dominated the competition, rather than relaxedly coasting to victory. However, every other team has other "what if" moments. We explore ours here purely out of curiosity.

* Mechanical Phish that never patched

To calculate our score in the absence of patching, we recalculated CFE scores, assuming that, any time an exploit would be launched on a CS against *any* team, the exploit would be run against us during that round and all subsequent rounds of that CS being in play. With this calculation, our score would be 267,065, which is 12,613 points higher than the patch strategy that we did choose and would put us in second place by a margin of over 5,000 points.

The prize for second place was $1,000,000. Patching at all cost us $250,000!

* Mechanical Phish that patched after attacking

Similar to the previous strategy, we calculated our score with a patch strategy that would delay patches until *after* we launched exploits on the corresponding CS. For the patches that would have been submitted, we used the same feedback that we received during the CFE itself. With this calculation, our score would be 271,506, which is 17,054 points higher than the patch strategy that we chose and would have put us in first place by a margin of over 1,500 points.

The prize for first place was $2,000,000. Patching stupidly cost us $1,250,000 and quite a bit of glory!

* Mechanical Phish that didn't do crap

We were curious: did we really have to push so hard and trade so much sanity away over the months leading up to the CGC? How would a team that did *nothing* do? That is, if a team connected and then ceased to play, would they fare better or worse than the other players? We ran a similar analysis to the "Never patch" strategy previously (i.e., we counted a CS as exploited for all rounds after its first exploitation against any teams), but this time removed any POV-provided points. In the CFE, this "Team NOP" would have scored 255,678 points, barely *beating* Shellphish and placing 3rd in the CGC.

To be fair, this score calculation does not take into account the fact that teams might have withheld exploits because all opponents were patched against them. However, ForAllSecure patched only 10 binaries, so it does not seem likely that many exploits were held back due to the presence of patches.

One way of looking at this is that we could have simply enjoyed life for a year, shown up to the CGC, and walked away with $750,000. Another way of looking at this is that, despite us following the worst possible strategy in regards to patching, the technical aspects of our CRS were good enough to compensate and keep us in the top three positions!

--[007.006 - Scoring Difficulties

Similarly to this being the first time that autonomous systems compete against each other in a no-humans-allowed match, this was also the first time that such a match was *hosted*. The organizing team was up against an astonishing amount of challenges, from hardware to software to politics, and they pulled off an amazing event.

However, as in any complex event, some issues are bound to arise. In this case, the problem was that measuring program performance is hard. We found this out while creating our CRS, and DARPA experienced this difficulty during the final event. Specifically, we noticed two anomalies in the scoring data: slight disparities in the initial scoring of challenge sets, and performance "cross-talk" between services.

* Initial CS Scoring

Patches can only be fielded on round 3 (after being submitted on round 2) of a binary being deployed. However we noticed that our availability scores were lower than our opponents, even on the *first* round of a challenge, when they could not yet be patched. In principle, these should all be the same, as a team has *no* way to influence this performance score. We calculated the average of the first-round CS availability scores, presented in the table below. The scores vary. The difference between the "luckiest" team, regarding their first-round CS score, and the "unluckiest"

team was 1.6 percentage points. Unfortunately, Shellphish was that unluckiest team.

Since the availability score was used as a multiplier for a team's total score, if the "luckiest" and "unluckiest" had their "luck" swapped, this would compensate for a total score difference of 3.2%. That is a bigger ratio than the difference between second and third place (2.9%), third and fourth place (1.1%), fourth and fifth place (1.7%), and fifth and sixth place (0.4%). The winner (Mayhem) could not have been unseated by these perturbations, but the rest of the playing field could have looked rather different.

Team	Average First Round Availability
CSDS	0.9985
ForAllSecure	0.9978
Disekt	0.9975
TECHx	0.9973
CodeJitsu	0.9971
DeepRed	0.9917
Shellphish	0.9824

* Scoring Cross-talk

We also noticed that performance measurements of one challenge seem to influence others. Specifically, when we patched the binary NRFIN_00066 on round 39, we saw the performance of *all* of our other, previously-patched, binaries drop drastically for rounds 40 and 41. This caused us to pull back patches for *all* of our patched binaries, suffering the resulting downtime and decrease in security.

Anecdotally, we spoke to two other teams, DeepRed and CodeJitsu, that were affected by such scoring cross-talk issues.

--[008 - Warez

We strongly believe in contributing back to the community. Shortly after qualifying for the Cyber Grand Challenge, we open-sourced our binary analysis engine, angr. Likewise, after the CGC final event, we have released our entire Cyber Reasoning System. The Mechanical Phish is open source, and we hope that others will learn from it and improve it with us.

Of course, the fact that we directly benefit from open-source software makes it quite easy for us to support open-source software. Specifically, the Mechanical Phish would not exist without amazing work done by a large number of developers throughout the years. We would like to acknowledge the non-obvious ones (i.e., of course we are all thankful for Linux and vim) here:

* AFL (lcamtuf.coredump.cx/afl) - AFL was used as the fuzzer of every single competitor in the Cyber Grand Challenge, including us. We all owe lcamtuf a great debt.
* PyPy (pypy.org) - PyPy JITed our crappy Python code, often increasing runtime by a factor of *5*.
* VEX (valgrind.org) - VEX, Valgrind's Intermediate Representation of binary code, provided an excellent base on which to build angr, our binary analysis engine.
* Z3 (github.com/Z3Prover/z3) - angr uses Z3 as its underlying constraint solver, allowing us to synthesize inputs to drive execution down specific paths.
* Boolector (fmv.jku.at/boolector) - The POVs produced by the Mechanical Phish required complex reasoning about the relation between input and output data. To reduce implementation effort, we wanted to use a constraint solver to handle these relationships. Because Z3 is too huge and complicated to include in a POV, we ported Boolector to the CGC platform and included it in every POV the Mechanical Phish threw.
* QEMU (qemu.org) - The heavy analyses that angr carries out makes it considerably slower than qemu, so we used qemu when we needed lightweight, but fast analyses (such as dynamic tracing).
* Unicorn Engine (www.unicorn-engine.org) - angr uses Unicorn Engine to

speed up its heavyweight analyses. Without Unicorn Engine, the number of exploits that the Mechanical Phish found would have undoubtedly been lower.
* Capstone Engine (www.capstone-engine.org) - We used Capstone Engine to augment VEX's analysis of x86, in cases when VEX did not provide enough details. This improved angr's CFG recovery, making our patching more reliable.
* Docker (docker.io) - The individual pieces of our infrastructure ran in Docker containers, making the components of the Mechanical Phish well-compartmentalized and easily upgradeable.
* Kubernetes (kubernetes.io) - The distribution of docker containers across our cluster, and the load-balancing and failover of resources, was handled by kubernetes. In our final setup, the Mechanical Phish was so resilient that it could probably continue to function in some form even if the rack was hit with a shotgun blast.
* Peewee (https://github.com/coleifer/peewee) - After an initial false start with a handcrafted HTTP API, we used Peewee as an ORM to our database.
* PostgreSQL (www.postgresql.org) - All of the data that the Mechanical Phish dealt with, from the binaries to the testcases to the metadata about crashes and exploits, was stored in a ridiculously-tuned and absurdly replicated Postgres database, ensuring speed and resilience.

As for the Mechanical Phish, this release is pretty huge, involving many components. This section serves as a place to collect them all for your reference. We split them into several categories:

--[008.001 - The angr Binary Analysis System

For completeness, we include the repositories of the angr project, which we open sourced after the CQE. However, we released several additional repositories after the CFE, so we list the whole project here.

* Claripy

Claripy is our data-model abstraction layer, allowing us to reason about data symbolically, concretely, or in exotic domains such as VSA. It is available at https://github.com/angr/claripy.

* CLE.

CLE is our binary loader, with support for many different binary formats. It is available at https://github.com/angr/cle.

* PyVEX.

PyVEX provides a Python interface to the VEX intermediate representation, allowing angr to support multiple architectures. It is available at https://github.com/angr/pyvex.

* SimuVEX.

SimuVEX is our state model, allowing us to handle requirements of different analyses. It is available at https://github.com/angr/simuvex.

* angr.

The full-program analysis layer, along with the user-facing API, lives in the angr repository. It is available at https://github.com/angr/angr.

* Tracer.

This is a collection of code to assist with concolic tracing in angr. It is available at https://github.com/angr/tracer.

* Fidget.

During the CQE, we used a patching method, called Fidget, that resized and rearranged stack frames to prevent vulnerabilities. It is available at https://github.com/angr/fidget.

* Function identifier.

We implemented testcase-based function identification, available at

https://github.com/angr/identifier.

* angrop.

Our ROP compiler, allowing us to exploit complex vulnerabilities, is
available at https://github.com/salls/angrop.

--[008.002 - Standalone Exploitation and Patching Tools

Some of the software developed for the CRS can be used outside of the
context of autonomous security competitions. As such, we have collected it
together in a separate place.

* Fuzzer.

We created a programmatic Python interface to AFL to allow us to use AFL as
a module, within or outside of the CRS. It is available at
https://github.com/shellphish/fuzzer.

* Driller.

Our symbolic-assisted fuzzer, which we used as the crash discovery
component of the CRS, is available at
https://github.com/shellphish/driller.

* Rex.

The automatic exploitation system of the CRS (and usable as a standalone
tool) is available at https://github.com/shellphish/rex.

* Patcherex.

Our automatic patching engine, which can also be used standalone, is
available at https://github.com/shellphish/patcherex.

--[008.003 - The Mechanical Phish Itself

We developed enormous amounts of code to create one of the world's first
autonomous security analysis systems. We gathered the code that is specific
to the Mechanical Phish under the mechaphish github namespace.

* Meister.

The core scheduling component for analysis tasks is at
https://github.com/mechaphish/meister.

* Ambassador.

The component that interacted with the CGC TI infrastructure is at
https://github.com/mechaphish/ambassador.

* Scriba.

The component that makes decisions on which POVs and RBs to submit is
available at https://github.com/mechaphish/scriba.

* Docker workers.

Most tasks were run inside docker containers. The glue code that launched
these tasks available at https://github.com/mechaphish/worker.

* VM workers.

Some tasks, such as final POV testing, was done in a virtual machine
running DECREE. The scaffolding to do this is available at
https://github.com/mechaphish/vm-workers.

* Farnsworth.

We used a central database as a data store, and used an ORM to access it.
The ORM models are available at https://github.com/mechaphish/farnsworth.

```
* POVSim.

We ran our POVs in a simulator before testing them on the CGC VM (as the
latter is a more expensive process). The simulator is available at
https://github.com/mechaphish/povsim.

* CGRex.

Used only during the CQE, we developed a targeted patching approach that
prevents binaries from crashing. It is available at
https://github.com/mechaphish/cgrex.

* Compilerex.

To aid in the compilation of CGC POVs, we collected a set of templates and
scripts, available at https://github.com/mechaphish/compilerex.

* Boolector.

We ported the Boolector SMT solver to the CGC platform so that we could
include it in our POVs. It is available at
https://github.com/mechaphish/cgc-boolector.

* Setup.

Our scripts for deploying the CRS are at
https://github.com/mechaphish/setup.

* Network dude.

The CRS component that retrieves network traffic from the TI server is at
https://github.com/mechaphish/network_dude.

* Patch performance tester.

Though it was not ultimately used in the CFE, because performance testing
is a very hard problem, our performance tester is at
https://github.com/mechaphish/patch_performance.

* Virtual competition.

We extended the provided mock API of the central server to be able to more
thoroughly exercise Mechanical Phish. Our extensions are available at
https://github.com/mechaphish/virtual-competitions.

* Colorguard.

Our Type-2 exploit approach, which uses an embedded constraint solver to
recover flag data, is available at
https://github.com/mechaphish/colorguard.

* MultiAFL.

We created a port of AFL that supports analyzing multi-CB challenge sets.
It is available at https://github.com/mechaphish/multiafl.

* Simulator.

To help plan our strategy, we wrote a simulation of the CGC. It is
available at https://github.com/mechaphish/simulator.

* POV Fuzzing.

In addition to Rex, we used a backup strategy of "POV Fuzzing", where a
crashing input would be fuzzed to determine relationships that could be
used to create a POV. These fuzzers are available at
https://github.com/mechaphish/pov_fuzzing.

* QEMU CGC port.

We ported QEMU to work on DECREE binaries. This port is available at
https://github.com/mechaphish/qemu-cgc.
```

```
--[ 009 - Looking Forward

Shellphish is a dynamic team, and we are always looking for the next
challenge. What is next? Even we might not know, but we can speculate in
this section!


* Limitations of the Mechanical Phish

The Mechanical Phish is a glorified research prototype, and significant
engineering work is needed to bring it to a point where it is usable in the
real world. Mostly, this takes the form of implementing the environment
model of operating systems other than DECREE. For example, the Mechanical
Phish can currently analyze, exploit, and patch Linux binaries, but only if
they stick to a very limited number of system calls.

We open-sourced the Mechanical Phish in the hopes that work like this can
live on after the CGC, and it is our sincere hope that the CRS continues to
evolve.


* Cyber Grand Challenge 2?

As soon as the Cyber Grand Challenge ended, there were discussions about
whether or not there would be a CGC2. Generally, DARPA tries to push
fundamental advances: they did the self-driving Grand Challenge more than
once years, but this seems to be because no teams won it the first time.
The fact that they have not done a self-driving Grand Challenge since
implies that DARPA is not in the business of running these huge
competitions just for the heck of it: they are trying to push research
forward.

In that sense, it would surprise us if there was a CGC2, on DECREE OS, with
the same format as it exists now. For such a game to happen, the community
would probably have to organize it themselves. With the reduced barrier to
entry (in the form of an open-sourced Mechanical Phish), such a competition
could be pretty interesting. Maybe after some more post-CGC recovery, we'll
look into it!

Of course, we can also sit back and see what ground-breaking concept DARPA
comes up with for another Grand Challenge. Maybe there'll be hacking in
that one as well.


* Shellphish Projects

The Mechanical Phish and angr are not Shellphish's only endeavors. We are
also very active in CTFs, and one thing to come out of this is the
development of various resources to help newbies to CTF. For example, we
have put together a "toolset" bundle to help get people started in security
with common security tools (github.com/zardus/ctf-tools), and, in the
middle of the CGC insanity, ran a series of hack meetings in the university
to teach people, by example, how to perform heap meta-data attacks
(github.com/shellphish/how2heap). We're continuing down that road, in fact.
Monitor our github for our next big thing!
`


--[ 010 - References

[Driller16] Driller: Augmenting Fuzzing Through Selective Symbolic
Execution
Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang,
Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna
Proceedings of the Network and Distributed System Security Symposium (NDSS)
San Diego, CA February 2016

[ArtOfWar16] (State of) The Art of War: Offensive Techniques in Binary
Analysis
Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario
Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser,
Christopher Kruegel, Giovanni Vigna
```

Proceedings of the IEEE Symposium on Security and Privacy San Jose, CA May 2016

[Ramblr17] Ramblr: Making Reassembly Great Again
Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, Giovanni Vigna
Proceedings of the Network and Distributed System Security Symposium (NDSS) San Diego, CA February 2017

[angr] http://angr.io

[Inversion] https://en.wikipedia.org/wiki/Sleep_inversion

[CGCFAQ] https://cgc.darpa.mil/CGC_FAQ.pdf