Xandra: An Autonomous Cyber Battle System for the Cyber Grand Challenge

Anh Nguyen-Tuong | University of Virginia
David Melski | GrammaTech
Jack W. Davidson, Michele Co, William Hawkins, and Jason D. Hiser | University of Virginia
Derek Morris | Microsoft
Ducson Nguyen and Eric Rizzi | GrammaTech

For the Cyber Grand Challenge, competitors built autonomous systems capable of playing in a capture-the-flag hacking competition. In this article, we describe the high-level strategies applied by our system, Xandra, their realization in Xandra's architecture, the synergistic interplay between offense and defense, and lessons learned via post-mortem analysis.

The challenge in DARPA's Cyber Grand Challenge (CGC) was to build an autonomous system capable of playing in a capture-the-flag hacking competition. The final event pitted the systems from seven finalists against each other, with each system attempting to defend its own network services while proving vulnerabilities ("capturing flags") in other systems' defended services.

The competition was organized around a collection of challenge sets (CSs), each consisting of one or more challenge binaries (CBs) that implement a network service. Each team in CGC developed a cyber reasoning system (CRS) that interacted with the competition framework (CF) to retrieve CBs; analyze the CBs to identify vulnerabilities; upload hardened replacement challenge binaries (RCBs); upload rule sets for a network intrusion defense system (IDS) to monitor and/or modify network traffic; upload proofs of vulnerability (PoVs) against competitors' services; and obtain feedback about fielded CBs, IDS rules, PoVs, and scores.

Each CRS was scored in each round of the competition for each CS live in that round:

- *Availability* measured the preservation of a CS's required functionality and overhead. It took values in the range 0–1. There were some allowances for overhead (20 percent file size, 5 percent memory, 5 percent performance), but steep penalties were imposed for exceeding that allowance or breaking functionality.
- *Security* measured the ability to block competitors' attempted PoVs. This score was 1 if any competitor succeeded in landing a PoV against a CS, and 2 otherwise.
- *Evaluation* measured the ability to land PoVs against competitors' services. It took values in the range 1–2 and was proportional to the number of competitors that were successfully PoV'd.

The total score for a CS in a round was computed as: availability × security × evaluation × 100. At the end

 Copublished by the IEEE Computer and Reliability Societies

of the game, each team was ranked based on the total score aggregated across all rounds and all CSs.

There were two primary mechanisms to defend a service: patching the CBs in a CS or installing IDS rules. Note that defending using either mechanism rendered the CS unavailable for one round, during which the score for the CS was 0. During this down round, replacement binaries or replacement IDS rules were made available to competitors for analysis.

There were two ways to prove a vulnerability: a Type 1 PoV represented a control flow hijacking attack and required demonstrable control of the instruction pointer and one other register. A Type 2 PoV represented an information leakage attack and required leaking and reporting four consecutive bytes from a designated memory page.

Scoring Tradeoffs

The scoring system precluded various strategies and favored others (see the DARPA Cyber Grand Challenge Competitor Portal for details: http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite).

For example, we initially considered a moving target defense strategy in which Xandra would replace binaries with a new, diversified variant every *n*th round, thereby forcing competitors to reanalyze our replacement binaries. The scoring penalty for fielding replacements made this strategy unworkable.

Getting a full score for security required defending against all competitors, making defenses that cover an entire attack class attractive. However, the strict performance and memory target envelopes of 5 percent favored strategies that were precise and that perturbed CBs only as needed to prevent a successful PoV.

On the offensive side, successful PoVs acted as bonus multipliers because there were no penalties associated with throwing PoVs against competitors. The initial rounds when a new CS was introduced by the competition framework was the only time a CS was guaranteed to be identical for all competitors. Thus the scoring rules provided strong incentives for finding PoVs quickly, before competitors could field defenses.

Xandra Strategy

The main elements of our strategy were:

- *Offense.* Use a combination of fuzzing and symbolic execution to find crashing inputs, and convert crashing inputs into PoVs.
- *Defense.* Defend at most once, and only when there is an indication that a CS might be vulnerable to a PoV: use control flow integrity (CFI) techniques to prevent Type 1 PoVs, use network filters to prevent Type 2 PoVs, and use point patching judiciously.
- *Safety.* Roll back any replaced binary with the original version when something goes wrong, for instance, if availability falls below a set threshold.

Our use of broad defenses provided the rationale for defending at most once (if a defense truly covers an attack class, there is no need to redefend). This strategic decision turned out to be crucial and well-adapted to the way the final game play unfolded.

However, broad defenses are generally costly in terms of memory and performance overheads. A primary challenge leading up to the final event was to develop lightweight defenses that retained the strong security property of covering an attack class while meeting the overhead constraints imposed by CGC.

Architecture

Xandra implements a virtual capture-the-flag team, with individual components that correspond to different roles. Figure 1 illustrates Xandra's architecture and workflow. At the core of the architecture is the GameMaster (GM), which is responsible for maintaining situational awareness; managing the work of other components; and submitting replacement binaries, IDS rules, and PoVs.

Upon detection of a new challenge set, the GM forwards the associated CBs to both offensive and defensive components simultaneously. In cases when a competitor uploads a new RCB, the RCB is forwarded only to the offensive component.

The offensive component is divided into two primary subcomponents: a fuzzing pod whose goal is to find inputs that result in a crash and a PoV generator that turns crashing inputs into PoVs.

The goal of the defensive component, Helix, is to generate functionally equivalent, protected, and efficient binaries. The default security policies are to armor binaries with CFI techniques to protect against Type 1 PoVs and to deploy a fixed set of IDS rules to handle Type 2 PoVs.

When a crashing input is found, the dynamic analysis engine (Daffy) generates point-patching policies and invokes Helix to reprotect binaries with the point patch applied.

Xandra uses the OpenStack cloud management infrastructure (https://www.openstack.org) to provision computing resources and to isolate components from one another. Xandra employs a bag-of-tasks model for both offensive and defensive components, using the Beanstalk work queue system (http://kr.github.io/beanstalkd). The advantages of this architecture are well-known: self–load balancing of tasks and natural fault-tolerance capabilities.

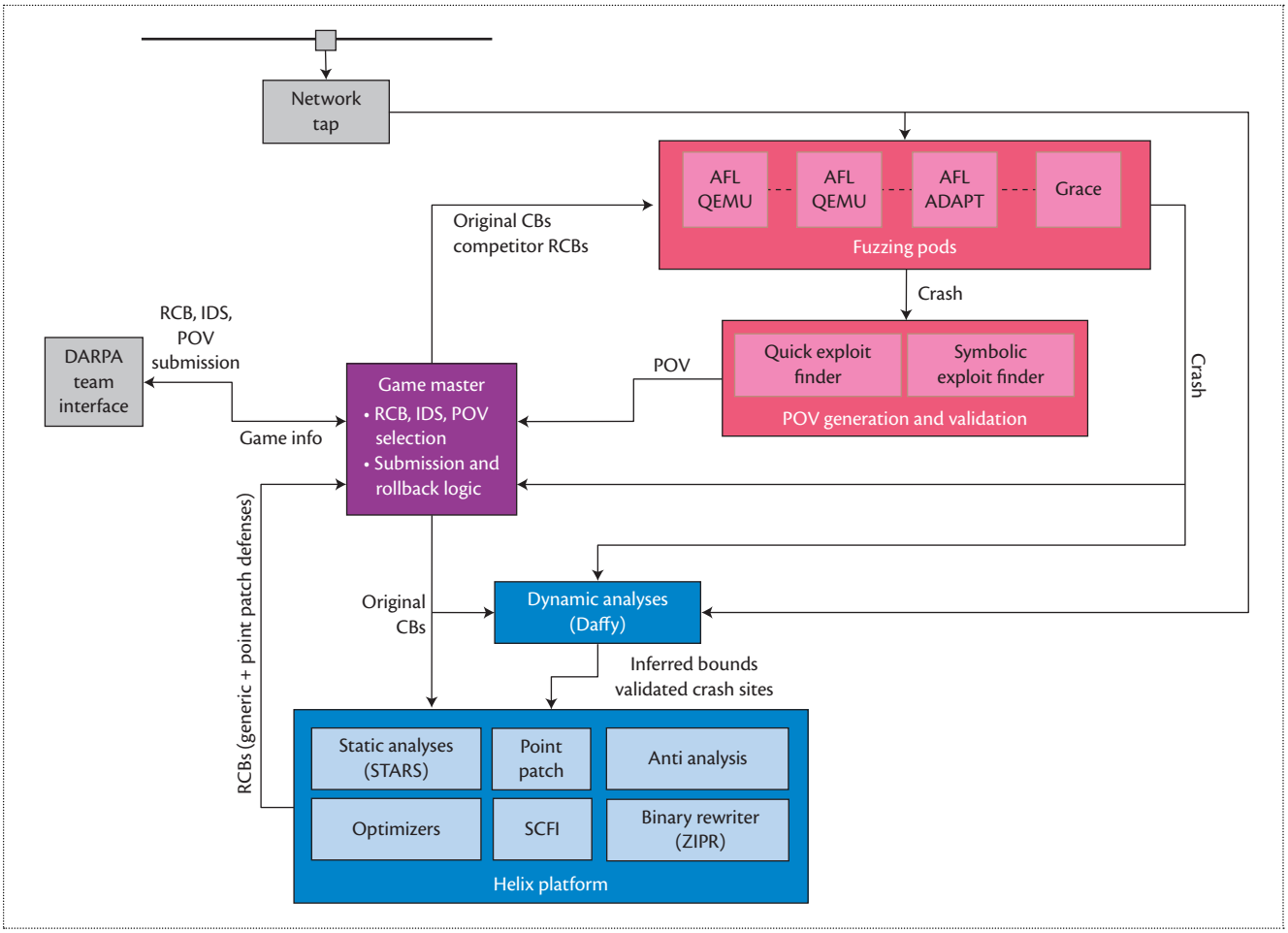

Figure 1. Xandra high-level architecture. The GameMaster is responsible for interacting with the competition framework (via the Team Interface) to carry out the game strategy by coordinating activities between offensive components (red) and defensive components (blue). Both offensive and defensive components had access to anonymized network traffic provided by the competition framework.

GameMaster

The GM manages the strategy described earlier and periodically polls the Team Interface for new information about the game state, for instance, current round number, new challenge binaries, replacement binaries and IDS rules fielded by competitors, and feedback on previously submitted binaries.

When Xandra successfully generates a new PoV, the GM immediately submits the PoV to the CF, provided that the reliability estimate for the PoV is higher than previous PoVs.

However, when Xandra generates a new protected binary, the GM waits to submit the binary until either of the following conditions is met:

- Xandra finds a crashing input.
- Feedback from the CF indicates that the original CS fielded has exited abnormally—that is, a competitor found a crashing input.

In other words, Xandra uses crashes as a proxy for exploitability.

When the GM decides to submit a replacement binary, it also submits a network filtering rule in the same round, provided the rule is unlikely to break functionality.

The GM performs a rollback if the CF reports broken functionality or if performance/memory overhead is higher than a somewhat arbitrary threshold of 30 percent.

Offense

Xandra uses both fuzzing, which (semi-) randomly generates test inputs, and symbolic execution, which generates and solves logical constraints to find inputs that lead to desired execution traces. Xandra's component for vulnerability discovery is called a *fuzzing pod*, even though it performs both fuzzing and symbolic execution. We provisioned sufficient fuzzing pods to analyze

30 CSs and RCBs from six competitors concurrently. Each fuzzing pod runs eight instances of the fuzzer and one instance of the symbolic execution engine. The fuzzing pod leverages captured inputs provided by the CF as seeds for the fuzzers, which can dramatically improve their effectiveness. Nondeterminism in a protocol, such as the use of nonces, can limit the utility of captured inputs as fuzzing seeds. Xandra uses a technique called *(de)noncification* to transform captured inputs into better seeds.

American Fuzzy Lop and extensions. Xandra's fuzzer is based on American Fuzzy Lop (AFL; http://lcamtuf.coredump.cx/afl), an industrial-grade fuzzer. AFL uses profiling to fingerprint the behavior exhibited by the subject program when run on an input. Inputs that lead to distinctive fingerprints (profiles) are given preference for further mutation and testing. AFL's performance is directly tied to how quickly it can generate profiles.

Xandra's fuzzing pods run a mixture of AFL instances in different configurations. Some instances use emulation (based on QEMU) to gather profiles, and some use binary instrumentation. The latter is typically 3x more efficient, but either technique can be foiled by antitamper measures. Xandra dynamically balanced the use of the two approaches on each CB based on observed effectiveness.

Fuzzing can be ineffective at triggering behavior that requires very specific inputs (for instance, "easter eggs," or hidden functionality).

We optimized AFL's profiling for use in CGC as follows:

- *Remove nondeterminism.* Nondeterminsm can lead to "noise" in AFL's profiles, which may negatively affect selection of inputs for mutation. Fortunately, the only source of nondeterminism in CBs was the random system call and the "secret page." We initialized these with fixed values during vulnerability discovery.
- *Skip transmit system calls that output to stdout or stderr.* AFL considers only the edge profile generated by executing on an input and ignores output by piping `stdout` and `stderr` to `/dev/null`. However, the program still incurs the overhead of placing the system call. We removed that overhead.
- *Skip unnecessary receive system calls.* A receive requesting zero bytes is treated as a no-op, and the system call is skipped.
- *Input delay heuristics.* When running a subject CB, AFL pipes the input to the CB's `stdin` file descriptor. Doing so means that the entirety of the input is immediately available to the CB. However, many CBs use a protocol that assumes there will be intermittent pauses in the delivery of the input. We implemented simple heuristics that model delays in the availability of input on `stdin` when the CB calls `fdwait`.
- *Skip sleep calls.* We developed simple heuristics to dramatically reduce sleep delays or skip sleeps entirely.
- *Late forking.* On Linux, launching a program requires a call to `fork()` to create a child process followed by a call to `exec()` to replace the child's image with the desired program. AFL modifies the subject program so that the `exec()` only needs to be done once to create an instance of the program that will act as a fork server. Thereafter, AFL alerts the server via a pipe to trigger forking of a child process that will continue execution on a new input from AFL, without the need for another call to `exec()`. By default, AFL places the fork server so it will be executed (and take control) before the subject program has done any computation. To avoid repeated and unnecessary execution of a CB's initialization code, we moved the fork server initialization to the first call to `receive()`.

Given the nondeterminism of fuzzing and the large number of CBs, we did not have sufficient resources for precise evaluation of the fuzzing pods. In a small set of timed experiments on 80 CBs from the CGC Qualifying Event, our AFL optimizations increased the number CBs crashed from 14 to 35.

Grace2 symbolic execution engine. AFL is effective at finding crashing inputs for the CSs in CGC, especially given seed inputs that demonstrate how to exercise the CS's functionality. However, fuzzing can be ineffective at triggering behavior that requires very specific inputs (for instance, "easter eggs," or hidden functionality). For example, fuzzing is unlikely to guess the one integer that would satisfy the condition (`user_int == MAGIC_CONSTANT`). In contrast, the constraint solvers used by symbolic execution have no trouble identifying an input that satisfies this condition.

Xandra augments AFL with symbolic execution to address this potential shortcoming with fuzzing. We built our symbolic execution engine, Grace2, to augment AFL. Its design is very similar to Driller,[1] the symbolic engine developed by Shellphish. In contrast to Driller, Grace2 immediately starts selecting inputs generated by AFL and attempts to mutate them to generate

inputs that will drive the program toward new, interesting states. Like Driller, Grace2 uses the edge profiles favored by AFL to determine whether or not an input is interesting.

PoV generation. Xandra employs redundancy for several of its capabilities. It uses two approaches to PoV generation: *Quick Exploit Finder* (QEF) and *Symbolic Exploit Finder* (SEF).

The first, QEF, uses inductive experimentation to determine if an input can be converted to a Type 1 or Type 2 exploit. It is simple, fast, and capable of working on any CS, including those that are multiprocess or tamper-proofed. However, it will succeed only for trivially exploitable vulnerabilities.

For Type 1 exploits, QEF runs the CS on a crashing input and looks for correlation between the register values at the crash site and the file contents. Wherever it finds a match, it mutates the appropriate file location and observes the effect (if any) on the register values if the program still crashes on the mutated input. If QEF found a way to control the IP register but not another register, it would attempt to find a portion of memory copied from the input and craft a code injection attack.

The search for Type 2 exploits begins during vulnerability discovery. The fuzzing pods monitor the CS output for the values we used to initialize the secret page and for simple transformations of those values. If a match is found, we experiment with modifications to the secret page to determine if we have a reliable Type 2 exploit.

QEF had the advantages of simplicity and efficiency. However, even minor transformations on the input could foil its ability to generate an exploit. To compensate, we developed the Symbolic Exploit Finder. SEF uses Grace2 to symbolically execute crashing inputs up to the crash site. At the site of the crash, it attempts to discover how much control it has over the registers and memory values involved in the crash, which helps it determine masks to use for Type 1 PoVs. To determine control of each register, SEF uses the symbolic expression for the register value. SEF uses a series of solver calls to identify the "fixed bits" that are the same in any execution. The remaining bits may still be constrained, though they differ in some executions. SEF performs a bounded search over candidate masks evaluating each candidate against randomly generated challenges, as the final PoV has to do.

If SEF determines it has sufficient control, it generates a set of constraints on the inputs that must be true for the input to meet the challenges produced by the CF. These constraints are then inserted into the SEF template PoV. The template PoV contains a copy of the Yices 2 SMT solver. Each template runs the script necessary for communicating with the CF during execution of a Type 1 PoV. At the appropriate place, it solves the constraints generated by SEF and feeds them to the subject binary. If a straightforward Type 1 exploit is not possible, SEF will also attempt an injection of a simple 12-byte program to set register values and a write-what-where attack to corrupt a pointer value.

During the competition, QEF was sufficient for the PoVs Xandra found.

PCAP Maximizer

Out of the box, AFL often struggles to achieve sufficient coverage on a given binary. One of the best ways to increase AFL's efficacy is to seed it with sample inputs that exercise various parts of the program. The CF provided this information in the form of PCAP files representing sample interactions with the programs. At a high level, our CRS captured this information and fed it into the each of the AFL instances running in the fuzzing pods. Just feeding these PCAP files naively into the various instances of AFL running under the fuzzing pod would have, in many cases, been suboptimal. To maximize the efficacy of these sample inputs, we did three things.

> **One of the best ways to increase AFL's efficacy is to seed it with sample inputs that exercise various parts of the program.**

Reduce PCAP file size. The sample interactions coming in from the CF were often too long; AFL does best with short inputs that can be run thousands of times per second. To shorten the inputs without losing unnecessary information, we reduced each PCAP file to the point that it would run through AFL within 25 ms. Like many parameters in the fuzzing pod, 25 ms was chosen based on AFL documentation and minimal experimentation.

Use AFL to iteratively noncify the program. The second complexity we had to address was the fact that each PCAP file provided by the CF was the output of a binary run under an unknown random seed. The use of random seeds meant that if there were any special tokens (nonces) in the program, playing back the same input on our own system would result in a different path being followed. For example, consider the program `KPRCA_00001`. This program gives the user a special key, based on some unknown random seed, that

must be input back into the program to access the main functionality of the binary. The input from this program looks something like this:

```
USER: Hello
PROGRAM: Your user ID is 648183
USER: 648183 says get /home/user
PROGRAM: Retrieving contents of /home/
user
...
```

Were we to naively rerun a sample input provided by the CF, we would almost certainly (because our random seed would be different) fail the playback test and not be able to exercise the same paths as the sample input. Instead, we would have gotten something like the following:

> **In contrast to most mutation strategies, noncification may use portions of a program's output (responses) to mutate the input.**

```
USER: Hello
PROGRAM: Your user ID is 715222
USER: 648183 says get /home/user
PROGRAM: Sorry, incorrect user ID,
goodbye!
```

To handle this problem, we leveraged AFL's profiles to "noncify" the input. That is, we looked for areas where the input and the output of the sample PCAP file matched. We considered these candidate nonces. We then tried various snippets of the sample PCAP input as nonces, each time running it through AFL to see whether the created variant resulted in a new path. We continued to do this in an iterative fashion, exploring each PCAP file (and its possible nonces) as deeply as possible.

Using AFL profiling, we could test multiple input variants very quickly, allowing us to handle cases where nonces were of an unknown length or ended with an unknown delimiter. In effect, we consider "noncification" to be a different mutation strategy that augments the mutation strategies used by AFL. In contrast to most mutation strategies, noncification may use portions of a program's output (responses) to mutate the input.

Focus on new interesting paths. We had to consider the large number of PCAP files that would be coming in from the CF. Given our understanding of the game specifications, we expected there to be hundreds of samples per minute. Given the number of inputs, it was likely that many of them would trace paths already explored by AFL. Therefore, we created one noncifier job specifically to handle the incoming PCAP file. After noncification, the noncified input would be distributed for further analysis by fuzzing and symbolic execution.

Defense

The Helix platform provided Xandra with the ability to transform binaries arbitrarily and efficiently. A key component of Helix is Zipr, an efficient static rewriter that transforms binary programs and libraries without access to their source code.[2] Zipr is compiler agnostic and applicable to both dynamically and statically linked programs and shared and static libraries. We have used Zipr to successfully rewrite such large code bases as libc, OpenJDK's libjvm, and the Apache webserver and its modules, and felt confident in its applicability to CGC binaries. The output of Zipr is a compact, transformed program or library with all the functionality of the original (unless that behavior is explicitly modified) and new functionality added through the application of composable user-specified transformations.

The primary transformations used by Xandra are block-level instruction layout randomization, selective CFI, point-patching techniques, various binary optimization techniques, and anti-analysis techniques. In addition to rewriting binaries, we also used network filters—all addressed below.

Block-level instruction location randomization (BILR). BILR is a diversity technique that randomizes the location of instructions by relocating code at the block level of granularity. Our original intent was to use BILR as a moving target defense to force competing teams to reanalyze and reattack our RCBs. However, the one-round penalty for submitting RCBs made this strategy prohibitively expensive. Instead, we used BILR as a defense-in-depth technique, coupled with selective control flow integrity (SCFI).

Selective control flow integrity. Control flow integrity techniques seek to ensure that control flow transfers adhere to a control flow graph specification.[3] To protect against Type 1 PoVs, we used a selective form of SCFI with the following characteristics:

- SCFI extracts the control flow graph directly from the binary.

- SCFI is coarse grained. All indirect control flow transfers—targets of indirect jumps, calls, and returns—belong to the same target class.
- SCFI is selective. We analyze binaries and look for safe indirect transfers as these need not be instrumented with CFI checks.

We classify functions as safe when we can prove that the return address for a given function cannot be overwritten. The proof enables us to skip the CFI instrumentation for returns from safe functions. We also forgo CFI instrumentations for switch tables. Omitting unnecessary CFI checks was key to obtaining good performance.

Because of the tight 5 percent constraint on memory and performance, we opted for a one-byte executable nonce, potentially at the risk of a larger attack surface.

Despite well-known weaknesses with coarse-grained CFI in general,[4] we felt that SCFI would be effective within the context of CGC—that is, it could be implemented efficiently and would raise the bar for competitors significantly.

Sample SCFI instrumentation is shown below:

```
(1) ...  ; at call to foo():
(2) call foo
(3) nop ; 1-byte executable nonce 0x90
(4)  ...    ; at return from foo():
(5) and [esp], 0x7FFFFFFF ; clamp
(6) mov ecx, DWORD [esp] ; assume ecx
    free
(7) cmp BYTE [ecx], 0x90 ; verify nonce
(8) jne _terminate
(9) ret
```

In this example, our instrumentation inserts a one-byte `nop` executable nonce after the call instruction (line 3). In function `foo()`, the return site is augmented to check for the presence of the executable nonce (lines 6–8). If an attacker is able to control the return address, the check for the executable nonce should fail. Because the stack is executable, we had to handle the case where an adversary may guess our executable nonce and insert malicious payload on the stack. To prevent control flow transfer to the stack, our CFI instrumentation clamps the return address (line 5) so transfers to code on the stack are not possible. We considered using a larger executable nonce, for example, two or more bytes, to increase the difficulty of finding return-oriented programming (ROP) gadgets. However, because of the tight 5 percent constraint on memory and performance, we opted for a one-byte executable nonce, potentially at the risk of a larger attack surface.

Daffy and point patching. Daffy is the name of Xandra's dynamic analysis engine. It is implemented on top of the Pin dynamic binary translator.[5] Daffy serves two primary purposes: bounds inference for stack arrays and generation of rules for faulting instructions. Daffy's approach was inspired by the dynamic reverse-engineering tool, Howard.[6] Daffy built up a database of memory access patterns based on traces of (presumed) benign, non-crashing inputs and contrasted them with accesses that caused a CB to crash. Based on the difference, Daffy inferred (possibly unsoundly) a memory access invariant. For a given CB, the output of Daffy is a set of faulting instructions along with policy rules to be applied (that is, bounds check or clamp).

In addition to broad-based defenses, Xandra also applies point patches for stack-based arrays. In cases where Daffy is able to infer array bounds, a bounds check is inserted. In cases where Daffy cannot infer bounds for the faulting instruction, a clamping technique is applied to instructions that read memory and load the value into a register. For example,

```
mov ecx, dword [eax]
```

when clamped, becomes

```
lea ecx, [eax]
and ecx, 0xbfffffff
mov ecx, dword [ecx]
```

The clamp value modifies only one bit and ensures that the memory read cannot originate from the flag page. The effect of clamping is to turn a Type 2 PoV attempt into a program crash.

Binary optimizations. To amortize the cost of our primary protections, we incorporated several optimization phases into the Helix tool chain. These included peephole optimization, register allocation, and CRCX eliding.

The first two techniques are standard compiler optimizations that were included because we noticed that many of the CBs used in the qualifying event and during the trial sparring sessions leading to the final event were not optimized (on some CBs, performance gains were substantial, up to 15–30 percent).

The third, *CRCX eliding*, was a CGC-specific technique. During testing, we noticed that DARPA added

an integrity check to all CBs, which we will call CRCX here. This code would run before the actual application code. Because this hash was immediately discarded after generation, this code had no real effect other than a performance and memory hit. To remove this code, we constructed a small tool that located points in the CB code where the CB had returned to the initial machine state but at a later execution point. The tool would update the entry point of the CB to be the later execution point. Subsequently, any CRCX code would be elided from the final binary as Zipr removes unreachable code blocks automatically.

Anti-analysis techniques. We developed three anti-emulation techniques, two of which targeted QEMU explicitly. To mitigate risks and overhead, we deployed only the two static techniques.

- *Zero page.* This technique exploits a bug wherein QEMU rejects binaries with headers that have zero for the size, mapped size, offset, and address.
- *String table.* This technique targets *objdump*, *gdb*, and similar programs. The key idea is that the kernel does not care about the section headers in the binary, so corrupting data related to them should only break analysis tools. We changed the listed number of sections to be 0, and the number referencing the index of the string table in the sections to be very large. These changes cause rejections or crashes in a number of standard analysis programs.
- *Floating-point divergence.* Our third technique was dynamic and relied on a floating-point divergence between the CGC operating system and QEMU for the `sinf` instruction. This divergence happened only for one specific value out of 2^{32} possible values.

We considered bootstrapping this last technique to crash the program, deceive symbolic engines into exploring decoy paths, induce infinite loops, and attempt to escape QEMU to interfere with competitors' CRSs. We decided against this course of action, because we already had a static, less expensive anti-QEMU technique and, most important, because we were not sure how far we could "push the limit" and stay within CGC rules. However, we could not rely on competitors to behave similarly. To mitigate the risk to Xandra, we took great care to isolate competitors' RCBs from critical components of Xandra.

Network defenses. Xandra has a single, fixed network filter rule set that it selectively deploys when it decides that defense is required and the rule set has a low probability of breaking functionality. Xandra's rule set is designed to block injections of addresses that reference the CGC secret page (in the range `0x4347C000` – `0x4347CFFF`). Xandra's rule set consists of 16 blocking rules, one for each possible value for the three high-order bytes on a secret-page address.

Xandra's rule set is prone to both false negatives and false positives. A false negative may occur if the injected address is encoded in any way. For example, a false positive would occur if the address is input as a decimal string and then converted by the targeted application. Despite this possibility, our hope was that the rules would block some attacks.

A false positive may occur if a secret-page address occurs (for instance, randomly) as part of benign input. To avoid false positives, Xandra checks for occurrences of secret-page addresses in the client-side network traffic that is captured in the round immediately after a

Table 1. CGC Final Event rankings.

CRS	Security	Evaluation	Availability
1. Mayhem	#6	#6	#1
2. Xandra	#1	#4	#2
3. Mechaphish	#2	#1	#5
4. Rubeus	#3	#3	#4
5. Galactica	#4	#2	#6
6. Jima	#7	#7	#3
7. Crspy	#5	#5	#7

Table 2. Net scoring defensive gains.*

CRS	Never PoV'd	PoV'd	Defensive gains
1. Mayhem	(477)	8,849	8,372
2. Xandra	(13,441)	15,071	1,630
3. Mechaphish	(25,308)	13,162	(12,146)
4. Rubeus	(10,901)	473	(10,429)
5. Galactica	(25,385)	8,188	(17,197)
6. Jima	(10,903)	244	(10,659)
7. Crspy	(27,971)	3,280	(24,690)

**Except for Mayhem and Xandra, cost of defenses on never-PoV'd CSs outweighed benefits of defending against PoV'd CSs.*

challenge set is released, before competitors have an opportunity to field PoVs. If Xandra detects any (apparent) secret-page addresses in this presumed benign input, the rule set is not deployed.

Postmortem Analysis

Table 1 provides the final rankings for each CRS. It is intriguing to note that Mayhem finished first despite being ranked #6 for both security and evaluation. Mayhem suffered a catastrophic failure that rendered it inoperable for one-third to half the event. However, by that time, Mayhem had built up a sufficient lead that it never relinquished.

Availability turned out to be the most important metric as only 20 out of 82 CSs (24 percent) were PoV'd. For the others, the optimal strategy would have been to leave the original CS alone and not attempt any defensive actions, as even a perfect replacement binary or IDS rule incurs a one-round availability penalty. While Xandra's high-level strategy to field an RCB at most once turned out to be a good fit, a key improvement for future competitions would be to refine Xandra's use of crashing as a proxy for exploitability and take into account the difficulty of turning crashes into PoVs.

Table 2 shows the total defensive gains realized by each team. Defensive gains or losses were computed by comparing availability and security scores against a no-defense policy. Given the ratio of PoV'd CSs, every team except Mayhem and Xandra could have improved their scores by employing a no-defense strategy!

Resiliency

One notable result during the competition was the importance of avoiding denial of service (DoS) through CRS overloads. Both Rubeus and Mechaphish had rounds where the availability scores plummeted for all (or most) CSs active in those rounds. This result implies they suffered essentially a DoS on the machine running their services during those rounds. Mayhem also seemed to suffer from a self-DoS from a bug that rendered their system inactive for a third to a half of the competition.

Xandra was carefully designed to minimize the chances of DoS. We were conservative in our application of defenses and our use of resources (Xandra drew the least power of any of the competitor systems). Possibly we did not maximally leverage the resources of our cluster, but we also avoided catastrophic failures.

Offense

Xandra succeeded in finding crashing inputs for many of the CSs and RCBs fielded by competitors. We suspect that some of the crashes, for instance, RCBs from Rubeus, were designed to derail analysis and were not actually exploitable. Considering just the original CSs, Xandra found crashing inputs from 38 of the 82 CSs, including 14 of the CSs that were proven vulnerable during the CGC Final Event (CFE).

Xandra's biggest weakness seemed to be converting crashing inputs into exploits. During the CFE, Xandra produced just four PoVs. In experiments after the CFE, Xandra regularly produced a PoV for a fifth CS, which demonstrates the nondeterminism of our fuzzing pods. After a bug fix, it was able to generate a PoV for a sixth CS. In addition, there were two more crashing inputs that gave control over the instruction pointer but not a general-purpose register. With a little more work on generating ROP exploits, Xandra would probably be able to generate PoVs for those as well.

Xandra made extremely good use of the vulnerabilities it did discover. Even though Xandra was sixth in terms of number of CSs proven vulnerable, it was fourth in its evaluation score. Many competitors seemed to focus on sophisticated vulnerability and exploit discovery. This strategy allowed them to find more exploits, but slowly. In contrast, QEF was very quick and, when it created an exploit, allowed the exploit to be thrown in more rounds.

Many competitors seemed to focus on sophisticated vulnerability and exploit discovery. This strategy allowed them to find more exploits, but slowly.

Defense

When deployed, Xandra's defenses were extremely effective during the competition. We found only two examples, one Type 1 and one Type 2, where a competitor's CRS managed to bypass Xandra's defenses.

Patching and IDS rules. Xandra replaced 42 of the CSs. On 26 of those, we applied only generic defensive policies. On 16, we also applied point patches.

Most RCBs had high availability (six were within 5 percent and 18 within 10 percent of the original) though three RCBs incurred severe penalties (more than 35 percent). Xandra reverted eight of the 42 RCBs it replaced (seven due to broken functionality, one due to exceeding our memory threshold). It also failed to revert two RCBs that should have been rolled back due to low availability scores but were not, due to a bug in the GameMaster logic.

Of the seven rollbacks caused by broken functionality, one was due to the rewriting process, another

due to control flow integrity, and yet another due to a CS performing an integrity check over its text and data segments. Four rollbacks were due to point patches that Xandra applied. An assumption behind the clamping policy is that it would be safe to clamp a faulting instruction. However, we should have realized that the faulting instruction could also access the flag page under benign inputs. A safer policy would have been to detect potential false positives, similar to what was done for the network filter rules, which we discussed earlier. Even then, our IDS rules also exhibited false positives.

We intended to add a module to Xandra that would use outputs from the PCAP maximizer to validate RCBs prior to deployment, but due to time and resource constraints, we settled for a much simpler validation process that consisted of testing the validity of RCBs using simple fixed inputs. This module would have dramatically increased Xandra's availability score.

Each cyber reasoning system in the competition had obvious flaws, many of which could be addressed easily. We surmise that if the competition were run again, perhaps with a different scoring system, the results might be quite different. In the end, Xandra performed admirably well, finishing second in the overall competition. Xandra was a bit overprotective, hardening services when it was not necessary. It also had a few bugs that caused it to break functionality. Despite these issues, Xandra was best in security, second best in balancing availability and security, and fourth in PoV generation.

The Cyber Grand Challenge provides a tantalizing glimpse into a future where vulnerabilities are discovered and remedied automatically by autonomous systems. Our team continues active research programs in reverse-engineering binaries, fuzzing, concolic execution, binary rewriting, and developing security transformations. Our ultimate goal is to scale CGC-developed techniques to real-world software and secure our cyber infrastructure. ■

Acknowledgments

We would like to thank our families, who provided unrelenting patience and support. Thank you to Kevin Scott and LinkedIn for donating a high-performance cluster. This material is based upon work supported by the Air Force Research Library (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-14-C-0110, FA8750-15-C-0118, and FA8750-15-2-0054. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFRL or DARPA.

References

1. N. Stephens et al., "Driller: Augmenting Fuzzing through Selective Symbolic Execution," *Proc. Network and Distributed System Security Symposium*, 2016.
2. W. Hawkins et al., "Zipr: Efficient Static Binary Rewriting for Security," *Proc. Dependable Systems and Networks*, 2017.
3. M. Abadi et al., "Control-Flow Integrity Principles, Implementations, and Applications," *ACM Transactions on Information and System Security* (TISSEC 09), vol. 13, no. 1, 2009, p. 4.
4. L. Davi et al., "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," *Proc. 23rd USENIX Conference on Security Symposium* (SEC 14), USENIX Association, 2014, pp. 401–416; http://dl.acm.org/citation.cfm?id=2671225.2671251.
5. C.-K. Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 05), ACM, 2005, pp. 190–200; http://doi.acm.org/10.1145/1065010.1065034.
6. A. Slowinska, T. Stancescu, and H. Bos, "Howard: A Dynamic Excavator for Reverse Engineering Data Structures," *Network and Distributed System Security Symposium*, 2011.

Anh Nguyen-Tuong is a principal scientist at the University of Virginia. Contact him at an7s@virginia.edu.

David Melski is a chief technology officer at GrammaTech. Contact him at melski@grammatech.com.

Jack W. Davidson is a professor of computer science at the University of Virginia. Contact him at jwd@virginia.edu.

Michele Co is a research scientist at the University of Virginia. Contact her at mc2zk@virginia.edu.

William Hawkins is a PhD student at the University of Virginia. Contact him at whh8b@virginia.edu.

Jason D. Hiser is a principal scientist at the University of Virginia. Contact him at hiser@virginia.edu.

Derek Morris is a software engineer at Microsoft. Contact him at dmorris321@gmail.com.

Ducson Nguyen is a software engineer at GrammaTech. Contact him at dnguyen@grammatech.com.

Eric Rizzi is a software engineer at GrammaTech. Contact him at erizzi@grammatech.com.