The Mayhem Cyber Reasoning System

Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, Alex Rebert, and Ned Williamson | ForAllSecure

Mayhem, one of the first generation of autonomous computer security bots that finds and fixes vulnerabilities without human intervention, won the DARPA Cyber Grand Challenge in August 2016. In this article, we detail Mayhem's creation and look forward to a future where autonomous bots will radically improve computer system security.

We are losing the battle against criminals who break into our computer systems. The reason: we rely only on humans to find new vulnerabilities and fix them. What if, however, we did not need to rely on human effort alone to find vulnerabilities? What if we could build intelligent computer bots that can find and fix vulnerabilities? And what if these intelligent bots could reduce the time to identify and remediate a vulnerability from human time scales—days, months, or years—to computer time scales of milliseconds?

The art of securing computer systems and processes against malicious actors has a broad frontier. DARPA's Cyber Grand Challenge (CGC) explored a new technology on the forefront of cybersecurity: the cyber reasoning system (CRS). A CRS is a fully autonomous system that takes complete responsibility for defending a set of software services. CRSs competing in the Cyber Grand Challenge demonstrated techniques in all core cybersecurity areas, including automatically developing firewall rules to stop attack traffic, analyzing programs to find bugs before an attacker, and patching vulnerabilities in compiled programs without any access to source code. Software defenses that might take human analysts hours, days, or weeks to develop and test were all deployed at machine speeds, on the order of tens of seconds, with no humans in the loop. This first generation of CRSs offers hope for an automatic first line of defense against attacks conducted using novel exploits on large scales at machine speeds, in addition to greatly tightening a defender's response time to other, more typical attacks.

We are the authors, designers, and developers of Mayhem, the winning CRS in the Cyber Grand Challenge, and this article discusses the design and implementation of Mayhem, lessons learned while preparing for the challenge, and our key takeaways from the competition.

DARPA's Cyber Grand Challenge

The CGC was a competitive, symmetric game played by seven fully autonomous CRSs and moderated by a "referee" scoring system. Here, we provide a brief overview of the game mechanics as well as the main ways in which CRS systems could attack and defend. We refer the interested reader to a presentation by the program manager[1] for a more comprehensive presentation of CGC.

Game Structure

The CGC was structured similarly to a networked "capture-the-flag" competition. Players raced to find, exploit, and fix software bugs in their services in an adversarial environment in real time. The competition started

 Copublished by the IEEE Computer and Reliability Societies

when the network was brought up. Each CRS was responsible for a networked server running an identical set of vulnerable software services. The referee served up previously unknown vulnerable software throughout the game.

The referee brokered all communications during the game. Each CRS was on a different and isolated network so that CRSs could not communicate with each other. In addition, a CRS did not have direct control over the server running the vulnerable software services and could only interface with that server through the referee.

To score points (or avoid losing them), players had to perform three main tasks during the competition. First, they had to protect their software from adversaries by finding and patching vulnerabilities. Second, they needed to keep their software available, functional, and efficient. Third, they needed to exploit vulnerabilities in their adversaries' software.

Players scored points by keeping their services running and exploiting the software of other teams, and lost points by damaging their own software or having their systems exploited. All exploit attempts were routed through the referee and mixed in with the referee's tests, which checked that patches to the defended software had not broken its functionality or performance. At the end of each five-minute round, the referee calculated the scores for all the players based on which services they kept running, how performant and functional those services were, and whose services were exploited by whom.

All CGC services were run on the DECREE architecture: a Linux variant with seven system calls and a custom executable format very similar to the binary format commonly used in Linux. This kept the scope of the engineering effort within the reach of small teams, while also keeping the platform close enough to Linux for tools to require straightforward engineering to extend to production systems. It also limited the damage teams could do to each other via exploitation.

Attacking

To attack a service, a CRS had to submit to the referee a program, called an *exploit*, that proved the existence of a security-critical software vulnerability in the target service. The submitted "proof of vulnerability" (PoV) program had to either gain code execution within the target service—which, in the real world, corresponds to taking over a server running that service—or manipulate the target service into leaking privileged data, a less versatile attack than hijacking a server but, as demonstrated by the infamous 2014 Heartbleed bug, a potentially serious vulnerability. Attacks against a particular service could be conducted up to 10 times per five-minute round. This meant that attacks did not need to be perfectly reliable, but an unreliable attack thrown many times increased the chances that the target would notice, patch their service, and reflect your exploit. Consequently, reliable and stealthy exploits were encouraged by the structure of the competition.

Defending

Each CRS had two independent methods to defend each of its services: intrusion detection and binary patching. To use the competition's intrusion detection system, CRSs had to write firewall rules in a language similar to the Snort Intrusion Detection System (IDS) rule language and submit them to the referee, which then deployed them in front of a given service. We did not use the IDS functionality, as its performance impact on the protected services was difficult to predict. Instead our defense relied entirely on binary patching—finding vulnerabilities in the services themselves and fixing them at the machine-code level. Typically, when vendors release a security patch, they make modifications to the program's source and recompile. We did not have the luxury of source, and consequently patching correctly was quite challenging.

A pivotal aspect of the CGC was that deployed firewall rules and patches were harshly penalized for disrupting the operation of services. This could occur either by misidentifying legitimate network traffic as an attack, introducing a bug into the service that caused it to malfunction, or degrading the performance of the service. The correctness and performance of a service were reflected by an "availability score" calculated by the CGC referee. Availability scores began to drop steeply at 5 percent overhead compared to an unpatched service with no firewall rules. Despite this high bar, after optimizing our patches for performance, most of the patches we deployed suffered negligible performance penalty: out of the seven patches that were scored by the referee, our average overhead was 2.64 percent for time and 0.98 percent for memory. All had perfect functionality.

In addition, to represent the cost of distributing a patch across a defended network, whenever a CRS deployed a patch or a network filter for a service, the referee took that CRS's copy of that service down for a round, effectively giving a 0 availability score for one round every time you patched. This made deploying a buggy patch very expensive—you lost a round of points when you deployed it, lost points every round that it was deployed, and then lost another round of points when you deployed a replacement patch.

These tradeoffs are not unique to the Cyber Grand Challenge. Network administrators commonly forgo applying security patches until they know the patches will not disrupt vital services. Vendors are careful to mention in security announcements how severe software vulnerabilities are as well as whether the bugs are known to be exploited in the wild so that their customers can decide how detrimental not patching would be. Today, this data is gathered by humans, but an automated solution able to

optimize service availability without sacrificing security would be immensely valuable.

Scoring

Because the scoring algorithm drove many of our design decisions, we briefly describe it here. For each service, the score was calculated each round for a given CRS as follows:

$$Score = Availability \times Security \times Evaluation$$

Availability is the measure of performance and functionality described above, security is 1 if any adversary proved a vulnerability in this CRS's instance of the service during this round and 2 otherwise, and evaluation is $1+\frac{x}{N-1}$ where x is the number of competitors successfully attacked on this service by the CRS, and N is the number of CRSs participating (which was seven).

The score for a given round is the sum of all the services' scores. Similarly, the total score of a CRS is the sum of its scores over all the rounds.

Mayhem Defense

Mayhem confirms and patches software flaws differently from a human developer or security analyst. When analyzing a service, Mayhem confirms a software flaw only when a test case that causes the service to crash or otherwise exhibit potentially exploitable behavior is available. Mayhem has no access to the original source code or the specification of the application (while both are typically available to the developer). Mayhem only knows how a service actually behaves and that services should not crash or be exploitable. Thus, patches deployed by Mayhem aim to render identified software flaws unexploitable.

Mayhem patches are based on runtime property checking. For every identified flaw, Mayhem inserts introspective checks into programs, similar to assertions that a developer might add. These checks verify runtime properties that are exceedingly likely to be true for a correctly operating C or C++ program, and not likely to be true for a CGC service that is being exploited. Specifically, a Mayhem patched service verifies at various points that it would not access memory at, or branch to, invalid or unusual addresses. Failing a check leads to safe termination of the service, preventing exploits from successfully completing.

Although these checks could be placed in thousands of locations throughout a service's code, too many such assertions reduces performance. Mayhem places checks along any code paths that it found crashing test cases for, and also at a few heuristically identified points. Furthermore, Mayhem uses formal methods to elide checks that are proven unnecessary.

Mayhem is also capable of defending its services using several established exploit mitigation techniques: stack canaries, address randomization, data execution prevention, and a simple tag-based control flow integrity (CFI) scheme. When utilized by a compiler, these mitigations can obviate entire classes of exploits, while rarely interfering with correct program behavior. Unfortunately, although introducing these mitigations into compiled services after the fact is possible, the added code may affect performance and/or functionality. Mayhem relies on its test case generation capabilities to verify that the inserted mitigations do not break the performance or functionality of services and to fall back to less comprehensive mitigation techniques when the performance overhead is unacceptable.

Binary patching is difficult. Even after deciding which checks and mitigations to insert where, Mayhem has to insert the patches and generate a functional, performant binary. Lacking source code, Mayhem patches services by directly editing the assembled processor instructions within the program. This is a nontrivial task because simply inserting instructions into the middle of a program would break the various fixed addresses and relative offsets with which a program refers to itself. This is similar to how inserting pages into a printed book would render the page numbers in the index and table of contents inaccurate, but much more critical and much harder to fix. In the next section, we discuss the core approaches Mayhem follows for tackling the practical issues of binary patching.

Binary Patching Techniques

Mayhem has two techniques for inserting patches (see Figure 1). Its preferred method, full-function rewriting (FFR), approaches the above challenge directly. FFR inserts patches into the middle of the program, and then attempts to repair the issues this causes by adjusting all addresses and offsets within the program accordingly. This is a tricky process, and our methods were not perfect: we relied fully on static analysis, which resulted in malfunctioning patched services on 0.57 percent of our test binaries. To ensure maximum reliability, we tested our methods daily on binaries released by the organizers and compiled across a variety of optimization levels during development, for a total of 293 test binaries. In addition, during the competition Mayhem was careful to not deploy FFR-patched binaries until it had verified that they functioned as intended on the referee's network traffic and generated test cases.

The other patching technique Mayhem employs is more conservative. Injection multipatching (IMP) avoids comprehensive modifications to programs by not inserting patches into the middle of a program. Instead, IMP substitutes a branch instruction into the program at each location to be patched. These branch instructions jump into a custom section appended to the end of the

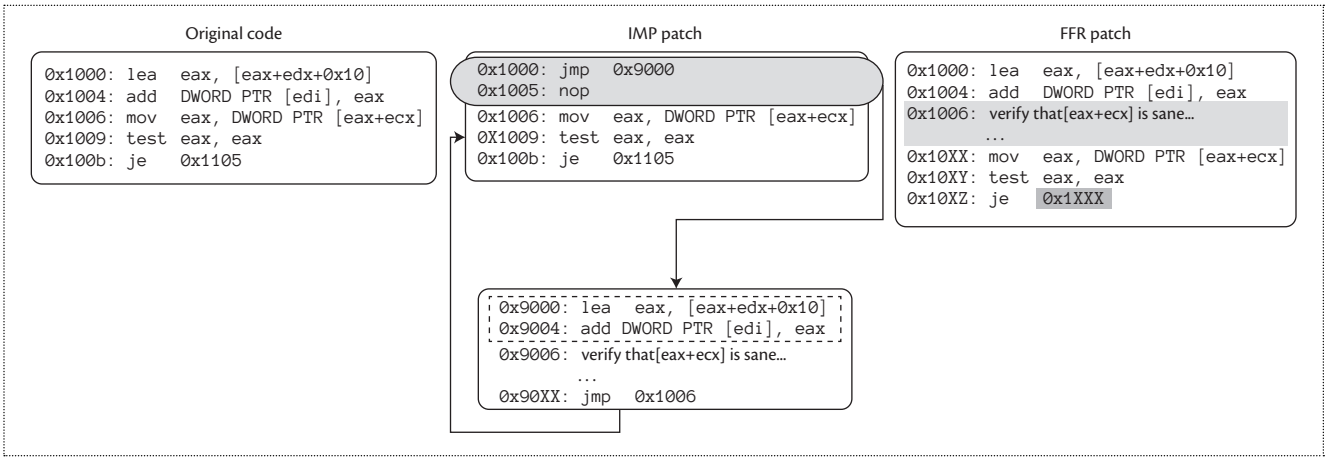

Figure 1. Injection multipatching (IMP) versus full-function rewriting (FFR) techniques. IMP clobbers some instructions to jump out to the patch body, located in a section at the end of the program. FFR inserts patches directly inline, but all program addresses are changed; many control flow instructions and code pointers must be updated accordingly.

program, which contains the instructions implementing the patch—along with any instructions displaced by the branch. To continue the book metaphor, this is like overwriting a sentence with "See Appendix A." IMP is more reliable than FFR because it leaves the majority of the program untouched, but the extra jumps to the added section increase the performance overhead of patched services.

Network Intrusion Detection

Mayhem did not use the network intrusion prevention system available to CRSs. Signature-based defenses like network filtering have trouble generalizing to defend against polymorphic exploit variants, and they might spuriously trigger on inputs that do not pose a threat to the service. Furthermore, they can inform other teams about the nature of the identified flaw. Mayhem's patches are much less susceptible to these failure modes, because they catch attacks as they exploit critical software flaws, do not interfere with normal service operation, and include obfuscation.

Mayhem Offense

Mayhem's offensive capabilities are based on generating test cases that demonstrate flaws in a target service. While humans tend to discover exploitable vulnerabilities by first identifying potentially flawed sections of code and then figuring out how to trigger that flaw, Mayhem needs to "see" a flaw occur before it can identify or attempt to exploit it. Mayhem makes up for this deficiency by generating and analyzing thousands of inputs per second, each of which could trigger a vulnerability.

Mayhem generates these test cases using a combination of gray box fuzzing and white box symbolic execution techniques. Other competing teams also combined the two.[2]

Symbolic Execution

Symbolic execution[3] is a method for reasoning about the behavior of a program by building logical formulae that represent the program. The inputs to the program are treated as variables, and any value computed by the program—such as return values, or branch conditionals that affect choices the program makes—is represented as a function on these input variables. Using an SMT solver, these functions may be solved for a desired output value to calculate an input test case that ensures the program will reach a desired state.

Mayhem's symbolic execution engine[4] manipulates the branch conditionals within a program. By generating inputs that will result in the program having different values for the conditionals it computes, Mayhem forces the program to execute in a variety of different ways. This exercises different areas of code within a program under a variety of circumstances, exposing software flaws.

The specific strategy Mayhem uses to generate test cases is known as *concolic execution*. Starting from an arbitrary initial seed test case, Mayhem traces the concrete (nonsymbolic) execution of the target program on that test case, while simultaneously building symbolic formulae for values derived from the input. Every time Mayhem encounters a conditional branch, it uses the formula for the condition to produce a modified input test case that takes the other direction on the branch. We call this production of modified test cases *forking*. Mayhem also forks for values other than branch conditionals, such as pointer values—this generates new test cases for which the program accesses memory differently. As Mayhem traces the seed test case, it forks many times. After the program completes execution of the seed test case, Mayhem repeats the tracing process using the forked test cases as new seeds.

Repeatedly tracing all forked test cases as new seeds leads to exponential growth of the number of seeds, rapidly overwhelming a system's capability to process them all. This "path explosion problem" is not specific to concolic execution; it arises from the fact that even simple programs tend to have an exponentially large number of possible states to explore. Mayhem employed a novel technique called Veritesting,[5] which mitigates path explosion by merging similar seeds together; processing a single merged seed achieves exploration equivalent to processing each of its constituent seeds.

During the CGC competition, our symbolic execution component found crashes for 62 of the 82 services, including five services that fuzzing did not crash.

Fuzzing

Our fuzzing was largely guided by the open source project American Fuzzy Lop (AFL; http://lcamtuf.coredump.cx/afl). This tool was designed to do intelligent, fast fuzzing using compile-time instrumentation to measure edge coverage in programs. Here, we describe some differences between AFL and our fuzzing.

As we did not have the source code for our target services, Mayhem uses alternative techniques for gathering coverage information. The first technique uses our FFR patching infrastructure: we inserted a series of patches that implemented the necessary code coverage tracking and reporting mechanisms. Although this technique was complicated and fragile, it allowed us to add many other useful features. For example, 32-bit comparisons were rewritten as four consecutive 8-bit comparisons to allow the fuzzer to make incremental progress with its random exploration and mutation, without needing to randomly guess the correct 32-bit value. In addition, initialization done by the binary could be skipped and analysis could be deferred until the program began reading user input, greatly accelerating fuzzing. In the end, FFR-assisted fuzzing allowed us to run the binary with negligible performance impact. In the rare event that FFR was not able to successfully patch the program, we fell back to a modified version of QEMU, as used by the AFL fuzzer. Although this was a safe option, it suffered a 100 to 900 percent performance hit.

During the qualifying event for the CGC, fuzzing helped us find bugs in 65 of the 131 binaries in 24 hours. Based on write-ups from other teams, this suggests that our fuzzer was far more effective than any fielded by any other team. Furthermore, continued improvements to our fuzzer between the qualifying and final events roughly doubled our bugs per hour; it found crashes in 92 of those 131 binaries from the qualifying event in a 16-hour period just before the final event. During the final CGC event, fuzzing found crashes in 62 of the 82 services (five of which were not found by symbolic execution).

Both of these techniques—symbolic execution and fuzzing—make use of code coverage metrics, particularly edge coverage. Coverage-guided program exploration searches for test cases that reach parts of the target program that no other test case has reached before. By analyzing the frontier of test cases that achieved new coverage, more and more parts of the code will be explored over time. By using the same coverage metric for test case prioritization in our fuzzer and symbolic executor, we minimized the amount of work duplicated between the two components. Working together, the two systems found crashes for 67 of the 82 services, just over 80 percent. Of the 57 crashes found by both components, symbolic execution found 27 before fuzzing, while fuzzing found 30 first.

Automatic Exploit Generation

After finding bugs, Mayhem had to exploit them. The Automatic Exploit Generation (AEG) component took in a CGC program (either an unpatched challenge binary or an opponent's patched binary), an opponent's network filter, and a crashing input and attempted to generate a PoV. To generate PoVs, we used a combination of black box and white box techniques. The black box AEG component inferred the relation between the input string and the crashing state through input mutations, and then generated an input that demonstrated the desired condition for a PoV (EIP control or a leak of sensitive data). The white box AEG component symbolically executed the crashing path and attempted to satisfy the constraints necessary for exploitation. If the constraints were satisfiable, Mayhem used the solution to emit a PoV. In addition, those constraints might depend on the data sent by the service. For instance, a service might expect a client to replay a nonce it sent or to solve a simple mathematical CAPTCHA. In those scenarios, the PoV generated the input on the fly by querying an embedded SMT solver.

All generated PoVs were tested in the DECREE environment to measure their reliability before deployment. At the end of the competition, Mayhem's database contained working PoVs for 18 services. This represents a lower bound on what we actually found, as our system deleted most (and sometimes all) PoVs for a challenge after that challenge was removed from the competition by the referee in order to free up disk space on the database node.

Mayhem Strategy

Strategy was an essential component. CRSs had to make strategic decisions throughout the game: Which binaries to patch? Which patches to deploy? Which teams to attack, and with which exploits? Where should limited resources be spent?

A bad strategy could ruin the winning chances of an otherwise good CRS. A single bad decision could have disastrous consequences. Indeed, as all the services were running on a single host, deploying one patched binary (or network filter) consuming too many resources could

affect all the services in play. In fact, this scenario happened to at least two CRSs, and might have cost one team third place and $750,000.

Patch Scoring and Selection

To avoid deploying bad patches, we took maximum advantage of the opportunities presented by the organizers to observe their scoring system and deduced which performance metrics were being used where the documentation was ambiguous. We also developed a distributed testing platform, which replayed seeds from the network against both our patched binaries and the unpatched versions to calculate our overheads and verify that we retained functionality. This allowed us to accurately predict how our patches would score before we deployed them, and we never fielded a functionality-breaking patch.

In addition to guaranteeing performance and functionality, we also had to balance uncertain security against known performance. Whereas our crash-specific patches were performant, they did not protect against vulnerabilities we had not yet found. Our binary hardening techniques could protect against unknown exploits, but they carried performance costs, and we often generated several variants with decreasing security before finding one with acceptable performance. In the end, we generated a priori estimates for the security scores of each type of patching and used these and our observed performance scores from testing to generate estimated scores for our patched binaries. When Mayhem decided to deploy a patch, it picked the one that it projected would score highest—a simple patch selection strategy, ignoring adversary behavior, but effective nonetheless. Given reasonably effective patches, choosing when to deploy was the critical part of the patching strategy.

Because the referee made deployed patches available to other CRSs during the competition, another option for patch selection, which we rejected, was "stealing" and redeploying other teams' deployed patches. Although this might be tempting if another CRS clearly had stronger patching capabilities, this exposes you to back doors, where an opponent could send your copy of his deployed patch a particular input and it could give him code execution. Because we added back doors to our patched binaries, we believed any adversary capable of patching better than us would probably do the same. Automatic back door removal would have required deep analysis of an adversary's patches, which might have exposed us to exploitation (as we discuss later), and so we did not consider it worth the risk. Because we never successfully exploited a back door against another team's service, we believe other teams reached the same conclusions about patch stealing.

Patching Strategies

Deploying patched binaries in the CGC was risky. Aside from the possibility of degraded availability, the patching process incurred one full round of downtime. Leaving binaries unpatched was dangerous, but patching a binary that was never going to be exploited was a straight loss of points. Deploying a poorly performing patch might also have been better in the long run than being constantly exploited. To address these nuances, we examined many possible patching strategies before finalizing on one.

Several naive patching strategies presented themselves, the most obvious of which was "always deploy as soon as you've generated a crash-agnostic patch you think has reasonable performance." Shellphish chose this strategy. Unfortunately, this strategy has some weaknesses. Patching everything as soon as possible led to large patching downtime penalties, and in several cases, Shellphish's patches performed worse than expected, leading to further penalties. In addition, some binaries were never exploited, so leaving the unpatched service up was optimal—as this avoided any downtime penalty and any risk of breaking functionality or performance. Even if a service was eventually exploited, if you generated and selected a good patch beforehand and deployed it promptly when exploitation began, you were only down one turn of exploitation points compared to the naive strategy. Detecting exploitation, however, was nontrivial due to the limited information available from the referee.

Although simple strategies are robust, we chose a complex strategy based on a Bayesian classifier. This allowed us to make informed inferences about which services were being exploited by other teams using all the data available to us. We used information including our points per round, what sort of test cases we had uncovered from our own bug hunting, and whether we had seen traffic crashing each service as observable values that were fed into a classification system with hard-coded initial Bayesian priors.

This system worked well for several reasons. Many of the relevant quantities to calculate the conditional probabilities were directly observable from the game state, giving us good sensors on which to base our measurements. We could also calculate many probability estimates in multiple independent ways and take the geometric average. This allowed us to turn several noisy probability estimates into a higher-accuracy estimate. Overall, this resulted in Mayhem being able to make many intelligent choices. For example, if teams attempted to trick our system into believing that it was under attack by sending "fake" exploits that crashed our service but did not actually exploit services, Mayhem would start treating crashing test cases as an unreliable indicator of being exploited (because our other sensors would not corroborate us getting exploited). This meant Mayhem could adapt to changing and unknown environments, which is a necessity in adversarial settings. The downside to this was that Mayhem's patching strategy was fundamentally reactive. Although it might detect quickly that it was being

exploited, it would never patch a service before it believed that there was an exploit in the wild.

Once we had an estimate for the probability that a service was being exploited, we compared it to a threshold to decide if we should deploy a patch. Rather than setting a fixed probability cutoff for when we should patch a service, Mayhem instead dynamically adjusted this threshold. This allowed us to adapt across a variety of possible strategic situations—if we observed that teams patching many or all services were doing well, we could become more aggressive in our patching to match them, but if we observed that high-scoring teams were conservative with their patching or that we were losing points for patching too aggressively, we could adjust this threshold.

Finally, once we determined that a service was probably under attack and we made the decision to patch, we evaluated the scores of our possible replacement binaries. In addition to our continuously updated performance estimates for our patches, we factored in estimates of how much longer the service would stay in the game and how badly a round of downtime would impact us (to avoid, for example, taking a round of downtime if the service was only expected to remain in play for one more round). After these calculations, if we decided that it would benefit our score to patch a service, we did so.

Anecdotally, we observed many cases in which Mayhem made "good decisions." We tested Mayhem's decision-making process through a number of practice CGC games and were often surprised by when it chose to patch or leave services unpatched. As humans we frequently make decisions irrationally, treating "worrying" indicators such as a crash in one of our services as strong indications of exploitation, even when statistical evidence in the game suggests these indicators are untrustworthy.

Offensive Strategy

CRSs had to make many offensive decisions throughout the competition. At each round and for each service in play, systems had to decide which teams to attack and which exploits to use. For instance, a CRS might not want to send an exploit to another system that is good at reflecting exploits by extracting them from the network tap.

Strategy is a function of the objective function. The CGC was no exception, and our strategy was specifically tailored to the CGC scoring. In particular, the scoring function penalized insecure binaries more than it rewarded successful attacks. Therefore, we concluded that the optimal offensive strategy was to send exploits, when available, to all teams at all times.

However, Mayhem still attempted to minimize reflection attacks by generating stealthy exploits. For instance, information disclosure attacks are harder to detect than a crashing input. Mayhem ranked exploits by stealthiness and submitted the stealthiest one.

Mayhem Autonomy and Counter-Autonomy

The CGC was a fully autonomous competition. No human intervention was allowed once the game started. The machines were entirely on their own, and the CRSs needed to maintain themselves for the duration of the competition in an adversarial environment.

One of our largest system resilience concerns was *counter-autonomy*, that is, adversaries sending us malicious input to disrupt the different components of our CRS (as opposed to exploiting the services provided by the referee). Here, we describe how Mayhem monitored itself to maintain uptime. We then discuss the different techniques that could be used to disrupt the correct operation of an autonomous CRS. Finally, we go over countermeasures to adversaries with counter-autonomy capabilities.

Autonomy

During development, we quickly realized that Mayhem would encounter scenarios that could not be predicted, because the CGC would be our first time handling inputs from real and motivated adversaries. Therefore, we worked under the assumption that components would go down. To simulate unexpected failures and challenge Mayhem's resiliency, we developed Chaos Monkey (https://github.com/Netflix/chaosmonkey).

Due to the full autonomy requirement, significant resources were spent on ensuring reliability and automatic recovery from failures. Our autonomy strategy relied on three main techniques: defensive programming, liveness monitoring, and redundancy.

We designed our components defensively. A significant portion of Mayhem's code consists of error handling, with a heavy use of timeouts to ensure progress. All components run in infinite loops with cron jobs to bring them back up if they die. We also have cron jobs to kill leaked processes and delete leaked temporary files.

To improve autonomy, we designed the Medic, a component for monitoring liveness and health of processes, VMs, and physical machines and restarting them as necessary. Liveness and health had different meanings for different components. For processes, that meant being alive and making meaningful progress, like testing new network inputs rather than wedging in a tight loop testing one pernicious input repeatedly. For VMs and physical machines, that meant that processes running on them were healthy and that CPU, memory, and disk usage were within expected bounds.

We added redundancy to many of our CRS components and ran them on multiple hosts concurrently. We ran multiple copies of the Medic monitoring each other and widely distributed our network seed testing, fuzzing, and exploit generation subsystems. For components that could not be duplicated easily, we designed failovers.

If one failed, the others would activate seamlessly. For instance, we put a lot of effort into securing our expected single point of failure: our database. We set up a hot/actively-synced replica of our database on another host, wrote automated failover routines, and tested this functionality extensively. We also had a failover for our patching system and our communications with the referee.

The only one of these reliability measures that we know fired during competition was the Medic, which killed the component downloading data from the referee to our database because it was running abnormally slowly. Unfortunately, killing that component did not solve the problem but made it worse, by increasing the backlog we had to pull down from the referee.

Counter-Autonomy

Counter-autonomy aims to hinder, deceive, or shut down an adversarial autonomous system's operations. In the CGC, CRSs could see traffic sent to their networked software as well as the patched binaries and network filters deployed by their adversaries. Those inputs should be considered malicious. Carefully crafted binaries or network packets could potentially slow down or take out an autonomous system for the rest of the competition. Counter-autonomy techniques were used by several teams during the CGC, and we discuss some of them here.

One of the least damaging counter-autonomy attacks simply slows down the target CRS. For instance, in the CGC, patched binaries from several teams[6] could detect when they were run under popular dynamic analysis frameworks like Intel PIN[7] or QEMU.[8] Under analysis, these binaries behaved differently, hindering the autonomous system by wedging the analysis system, wasting CPU cycles in tight loops, or allocating gigabytes of memory. Deploying patched binaries that varied their behavior when run in different environments was somewhat risky, as we had limited guarantees about the environment in which the binaries' performance would be measured for scoring purposes. Nevertheless, some teams considered the risks worth the potential benefits.

More damaging counter-autonomy techniques attempt to destroy rather than slow down the adversary's system. The goal of this sort of attack is to directly exploit components of the autonomous system itself. For instance, during the CGC, we assumed that most CRSs would replay packets received from the network tap against the unpatched version of the binary to detect crashes and exploits. On a crash, CRSs would likely perform additional analysis under a dynamic analysis framework. A malicious test case could exploit the unpatched binary to gain code execution, and then exploit the dynamic analysis framework to run code directly on the machine doing analysis. If no countermeasures were in place, these attacks were potentially game-ending. For example, a particularly nefarious payload could connect to the unauthenticated power API to turn off the adversary's machines for the rest of the game. Another option was to scrape the disk for credentials to the referee API and submit broken patched binaries for all the services in play, preventing the target from scoring any points. Although the CGC rules prohibited attacks of this kind, as security researchers, we planned for the worst. We took precautions to prevent these attacks on our system and to minimize our risk.

Finally, more elaborate and subtle counter-autonomy techniques involve deception in order to trick adversaries into making costly strategic mistakes. These are perhaps the most difficult to defend against. Attacks of this kind require predicting how adversaries will react to certain behaviors and using those reactions to your benefit. For instance, we predicted that once a service started crashing, a CRS might infer, perhaps incorrectly, that the service was being exploited, and consequently patch it. Due to the scoring algorithm, patching too early was a net loss; if no team was exploiting you, deploying a patch meant taking a performance and downtime penalty for no gain. Therefore, Mayhem sent harmless crashes like null-pointer dereferences to services we could not exploit, and it led some CRSs to make strategic mistakes in their patching decisions during the competition. In general, however, the structure of the competition made deploying deception difficult, as we had no data on the strategy and behaviors of our opponents. In a multicompetition series, however, deception would become important.

Counter-Counter-Autonomy

An autonomous CRS requires counter-autonomy countermeasures to be resilient in an adversarial environment. Consequently, we designed several protections in Mayhem. At a high level, our countermeasures are split into three categories: minimizing the attack vectors, fault isolation, and sandboxing.

Mayhem minimizes analyses conducted on attacker-controlled inputs. In particular, Mayhem never leverages static analysis and limits use of dynamic instrumentation frameworks on an adversary's patched binary. This was a conscious design decision: we did not think the benefits would outweigh the risks. On the one hand, this handicapped our CRS, as Mayhem could potentially have found more exploits by analyzing adversaries' patches. On the other hand, doing so would have put us at a greater risk of being exploited and taken down.

Mayhem did, however, run adversaries' patched binaries (albeit without analysis) to check that our exploits worked locally before deploying them to the network. This was a calculated risk, but one that we minimized with isolation. To limit the damage from a potential attack, Mayhem had one dedicated virtual machine per adversary, and it was the only machine on which we ever ran

their binaries. The effects of an attack would be limited to that one VM (assuming no ability to escape from the VM) and would not affect our analysis of other teams or the unpatched services.

Finally, we needed to consider malicious test cases from the network. Network traffic contained test cases sent by the referee to check availability, and those test cases were critical for both our offense and our defenses—they provided initial seeds for our fuzzing and symbolic execution, and they were also ground-truth for the intended functionality of each service. However, this network traffic potentially included malicious traffic from adversaries. Therefore, we handled data from the network (as well as data derived from it) with care. We hardened our dynamic analysis frameworks against public and homemade exploits, and we sandboxed our analysis processes with a combination of system call whitelists and limits on CPU and memory.

Mayhem's Autonomy in the CGC

Despite all the techniques mentioned above, we are sad to say that Mayhem still ran into issues during the competition. We failed to account for a failure mode in the component downloading round data from the referee. Although downloading a single round usually took less than 30 seconds, it got much slower starting at round 27 (out of 96). Some rounds took more than 18 minutes to download fully. As rounds were only five minutes long, this component therefore fell behind, and Mayhem did not have access to the current state of the game. Our CRS was essentially playing the game in the past: it was analyzing binaries after the referee had removed those services from the game, unaware that new services had replaced them.

Luckily for us, the number of exploits thrown by other teams decreased significantly starting around the time this issue occurred, and therefore Mayhem's score was not as negatively affected as expected. This lack of exploits made leaving the unpatched binaries up a surprisingly good strategy; it avoided downtime and performance losses. As a result, Mayhem held onto its lead from the first 27 rounds and still won first place. Interestingly, Mayhem recovered near the end the game and landed an exploit in the last round.

The Cyber Grand Challenge shows hope for technologies that were recently considered pure science fiction. With some of the top researchers in academia and industry working for two years toward a goal of automating large parts of software and network security, many hurdles have been overcome. ForAllSecure has always believed that these technologies would transfer to real-world applications, and we are excited to continue development to make automated security practical for real networks and software applications. Automated bug finding, patching, and software testing will help scale up security, hopefully allowing us to catch up to the blistering pace of modern technological development and ensure a safe computing landscape for everyone. ■

References

1. M. Walker, "Machine vs. Machine: Lessons from the First Year of Cyber Grand Challenge," *Proceedings of the 24th USENIX Security Symposium*, 2015.
2. N. Stephens et al., "Driller: Augmenting Fuzzing through Selective Symbolic Execution," *23nd Annual Network and Distributed System Security Symposium* (NDSS 16), 2016.
3. E.J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.
4. S.K. Cha et al., "Unleashing Mayhem on Binary Code," *Proceedings of the IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.
5. T. Avgerinos et al., "Enhancing Symbolic Execution with Veritesting," *ICSE*, 2014.
6. "Cyber Grand Shellphish," Shellphish, phrack.org, 2017; www.phrack.org/papers/cyber_grand_shellphish.html.
7. C.-K. Luk et al., "PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.
8. F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," *Proceedings of USENIX Annual Technical Conference*, 2005.

Thanassis Avgerinos is a cofounder of ForAllSecure. Contact at thanassis@forallsecure.com.

David Brumley is the CEO and cofounder of ForAllSecure and a professor at Carnegie Mellon University in ECE and CS. Contact at dbrumley@forallsecure.com.

John Davis is a software engineer at ForAllSecure. Contact at jedavis@forallsecure.com.

Ryan Goulden is an engineer at ForAllSecure. Contact at ryan@forallsecure.com.

Tyler Nighswander is a bidirectional engineer at ForAllSecure. Contact at tylerni7@forallsecure.com.

Alex Rebert is a computer security researcher and cofounder of ForAllSecure. Contact at alex@forallsecure.com.

Ned Williamson is a software engineer at ForAllSecure. Contact at ned@forallsecure.com.