Jeanne-Marie Musca
COMP 116
Final Project

The Cyber Grand Challenge
Autonomous detection and patching of online vulnerabilities

Abstract

Early this August, the finals of DARPA's "Cyber Grand Challenge" (CGC) were held, in which autonomous computer programs competed against each other in a "Capture the Flag" challenge. These programs had to find vulnerabilities in software they were given, fix these flaws, and try to exploit flaws in the software given to other programs. We explore the inaugural CGC from the perspective of both those competing and those running the CGC.

Introduction

In late October of 2013, DARPA announced that they intended to hold the "Cyber Grand Challenge (CGC) – the first-ever tournament for fully automatic network defense systems" [1]. The summer of 2016, two and half years later, seven teams competed for millions of dollars in the finals of the inaugural CGC. They watched on the sidelines as their autonomous systems were put to the test, each trying to protect a server and attack the servers protected by the other six systems. So, what goes into building one of these autonomous systems? Moreover, how do we evaluate the capabilities of such systems? These are the two questions that I endeavour to answer in this paper. However, before considering the specifics of the autonomous systems developed and of the event itself, let's take a step back and consider the motivations behind such a challenge.

I. To the Community: Why Autonomous Systems for Network Protection?

Broadly speaking, I am using the term "autonomous systems" to refer to a specific class of systems, specifically, CRSs (cyber reasoning systems). DARPA defines CRSs as "unmanned systems that autonomously reason about novel program flaws, prove the existence of flaws in networked applications, and formulate effective defenses" [2]. In essence, CRSs are stand-alone systems that can both attack and protect networks without relying on any human intervention.

Clearly, given the large amount of money invested in CGC, DARPA believes that autonomous systems should play an important role in defending networks against attacks [3]. Their position is understandable given the proliferation of vulnerabilities in the software currently used on networks, and the fact that human error is responsible for introducing these vulnerabilities. As Tim Rain, Director of Security at Microsoft, puts it: "As long as *human beings* write software code, mistakes that lead to imperfections in software will be made…. The challenge of finding vulnerabilities in very complex code is compounded by the fact that *there are an infinite number of ways that developers can make coding errors* that can create vulnerabilities, some of which are very, very subtle." [emphasis mine] [4]. Heartbleed, for instance, is a now classic example of how difficult it is for humans to detect errors generated by other humans, given that it went undetected for years [5]. In short, we humans are really good at creating software that contains vulnerabilities, and, these vulnerabilities can be very difficult for other humans to detect.

Of course, the problem is not that the vulnerabilities exist, per se. Rather, the problem is that many of these vulnerabilities *can* be detected, and that, chances are, the person who *does* find one of them will use it to attack a network, rather than protect it. So, in addition to detecting these vulnerabilities, patches should be quickly created and implemented in order to prevent

exploitation. However, we find that after vulnerabilities are discovered, patches may not be immediately issued by software companies [6] [7], and that once patches are issued, the end users may not apply them [8]. So, the problem with software vulnerabilities on networks is not just detecting them, but also fixing them once they are discovered.

Autonomous systems like those created to compete in the CGC are designed to address both these problems at once. They detect vulnerabilities, create patches and apply these patches. Such systems could be placed at several places in the lifetime of software. Before release, software could be examined by the systems for vulnerabilities, and these very same systems could fix problems as they found them. Furthermore, network administrators could use autonomous systems to detect and fix vulnerabilities in the software already deployed on their network. Presumably, these systems could analyze more code, more thoroughly than any human could, ultimately making our networks safer than they currently are.

Some considerations, however, might tamper our enthusiasm. First, we may not be able to control who gets access to these autonomous systems. The very design of the CGC as a capture the flag challenge illustrates the fact that attackers will find such systems useful. A system that can find a vulnerability and patch it, can also be used to exploit the detected vulnerability. Secondly, if companies don't have the incentive to release patches for the vulnerabilities they are aware of, they may not have the incentive to invest in these autonomous systems either. So, while the potential exists for autonomous systems to make networks much safer than they currently are, their simple existence does not guarantee a more secure network; they have to be put in the right hands. Keeping both the great potential for and possible risk of these autonomous systems in mind, we now turn to one of these autonomous systems.

II. Meet a CRS: Mayhem

Although seven autonomous systems made it to the CGC finals, we will focus on Mayhem, the winner**.** Mayhem was built by "ForAllSecure", a team with strong connections to Carnegie-Mellon University through its founders, David Brunley (faculty member at CMU), Thanassis Avgerinos and Alex Rebert (both grad students at CMU). ForAllSecure outlines its general strategies in a blog that it started while preparing for the CGC [9][1]. To get an idea of the diversity of techniques used in Mayhem, Table 1 shows how ForAllSecure dealt with different aspects of the challenge, as detailed in their blog.

Table 1: Techniques used to create Mayhem

Offense (Bug Detection and Exploitation)	**Symbolic Execution**
	Directed Fuzzing
Defense (Patching)	**Hot-patching**
	Recompilation

Mayhem had to both go on the offensive by finding and exploiting bugs in the opponents' servers, and defensively patch bugs on its own server. These different tasks required turning to several different algorithms. The ForAllSecure blog itself does not contain a detailed account of the actual algorithms driving Mayhem. Still, we can refer to recent academic papers and patent applications written by the members of ForAllSecure to get an idea of how they implemented Mayhem. For the sake of brevity, we will only give a detailed account of the offensive side of Mayhem's implementation. By doing so, we can see how ForAllSecure combined two techniques (Symbolic Execution and directed fuzzing) to achieve their winning results.

[1] Also, for both entertaining and informative tweets take a look at Mayhem's twitter account: @MayhemCRS

i. Symbolic Execution

"Symbolic Execution" is designed to be an intermediate between program *proving* and *testing*[2] (that is, between *static* and *dynamic* analysis). It merges these two approaches by:

1) defining symbolic representations that capture classes of possible input to a given program,
2) augmenting the original code of the program so that it accepts and outputs this symbolic representation (generally done by wrapping the original code in other code),
3) specifying the expected symbolic output, in response to a symbolic input and, finally,
4) running the augmented program to verify that the actual output matches the expected output. [10]

This technique, therefore, merges the static and dynamic approaches by proving general properties about the behaviour of a program, like static proofs, but doing so by running the actual code, as with dynamic testing.

Avgerinos et al. (founders of ForAllSecure) present a particular version of Symbolic Execution. This version leverages two ways in which the original code can be modified to interact with symbolic representations of input [11] [12][3]. Specifically, the authors employ a combination of what is called Static Symbolic Execution (SSE) and Dynamic Symbolic Execution (DSE). In broad strokes, SSE represents the whole program as a formula, and DSE translates parts of the program as it is being run, that is, as the input is following specific paths through the code.

---

[2] Program *proving* involves a static examination of the code. The person proving properties about a program must characterize the behaviour of the program using logical rules, and prove properties about these rules. The proof, then, is only as good as how well the rules capture the actual behavior of the program and how rigorous the prover was in generating a given proof. On the other hand, program *testing* involves running the program on a variety of inputs, and documenting the results. In this case, the tester can only be confident that the code will behave as expected for input similar to that contained in the test input. [10]

[3] An earlier version of this algorithm is also described here: [20].

These two methods have complementary strengths and weaknesses. DSE is generally quick, because it only has to deal with small parts of the code at a time. However, places where the code branches in response to the symbolic input can lead to "path explosion". Avgerinos et al. provide the following example code to illustrate the point:

```
int counter = 0, values = 0;
for (i = 0; i < 100; i ++) {
     if (input[i] == 'b') {
          counter ++;
     values +=2
     }}
if (counter == 75) bug();
```

Seeing as a character can take two paths in this code, DSE analysis will lead to 2^{100} different branches. Exploring this space is unfeasible. SSE, in contrast, can quickly summarize the effects of both branches. However, SSE analysis does not scale up well to large programs given that it processes the entire code at once. So, Avgerinos et al. propose an algorithm that uses the strength of both DSE and SSE. Initially, the algorithm uses DSE. However, if a branch is encountered, a version of SSE is called to analyze just that portion of the code. Thus, the problem of path explosion is avoided, and SSE is only ever called on small portions of the code.

ii. Directed Fuzzing

In addition to Symbolic Execution, Mayhem uses directed fuzzing. Directed fuzzing involves generating input that is likely to trigger vulnerabilities. Again, we don't yet have access to the specific algorithm driving Mayhem, but we can take a look at what members ForAllSecure have written about directed fuzzing recently: specifically, a paper on "Optimizing Seed Selection for Fuzzing", by Rebert et al. [13].

First, they present a general algorithm for fuzzing:

1) Discover the command line arguments to a program that allow a file to be read in.
2) Determine the file type(s) expected by an application (eg. pdfs).

3) Select a subset from a set of seeds (eg. all the pdfs found by an internet search).
4) Use the seeds to fuzz the program and uncover bugs.

Although Mayhem automates all these steps, the paper in question, though, focusses on step (3), choosing a subset of seeds. The selection of a subset is made necessary by the fact that a set of seeds may be huge. Thus, Rebert et al. are interested in producing formal rules for this selection process. One strategy is to select the subset randomly. However, many members of the set may follow similar paths in the code, so, a more efficient strategy could be to select the members that are most likely to follow different paths.

Rebert et al. examine the best algorithm for choosing seed files. Their tests found that some seed selection algorithms outperform random seed selection. Specifically, an algorithm called *Minset* outperforms the rest. Minset searches for a minimal set cover of the set of all seeds. A *set cover* of a set X is a subset of X, Y, in which all the elements that appear in X also appear at least once in Y. The *minimal* set cover is the set cover that contains the least sets. In terms of selecting a set of seeds for fuzzing, an element of the set is the part of the code that will be executed in response to the seed. Thus, looking for the minimal set cover involves looking for the smallest number of seeds that will cover the most code. Given Rebert et al.'s results, ForAllSecure likely used a similar algorithm to generate their fuzzing seeds.

iii. Why two techniques for detecting and exploiting bugs?

Symbolic Execution and directed fuzzing work well together while attacking other systems. ForAllSecure found that their fuzzer was good at finding simple cases, but did not pick up on the more complex vulnerabilities. The symbolic executor, on the other hand, was able to find complex paths in the system, but was slow at exploring these paths, and exploiting potential vulnerabilities. So, the symbolic executor would find interesting pieces of code which it would

then send to the fuzzer. The fuzzer would then be able to target that specific section with inputs and hopefully crash the system. [9] Thus, we see that an important part of designing an autonomous system is understanding how diverse algorithms complement each other, and how they can be coordinated.

Needless to say, there are many more interesting aspects of Mayhem to explore, such as its patching algorithms. Still, we leave that exploration for a future paper and now turn to the creation of the CGC itself, the challenge that fostered the development of autonomous systems like Mayhem.

III. How DARPA created the CGC

As mentioned, two and a half years elapsed between the announcement of the Cyber Grand Challenge and the finals that were held on August 4, 2016. During those years DARPA finessed their methods for measuring the success of the autonomous systems, identified and funded strong contenders, provided a testing infrastructure for the competitors and, ultimately, winnowed down the competition to the seven finalists. We will focus on how they evaluated the contenders, and on the software infrastructure that was created to run the challenge.

i. Evaluating the CRSs

In the course of evaluating the CRSs, DARPA hosted the CGC qualifying event (CQE) where all teams from both the proposal track and open track demonstrated the capabilities of their autonomous systems. From the CQE, teams were selected to be in the CGC finals (CFE), at which point, all teams would be funded. DARPA outlined 5 Areas of Excellence (AoE) that would be tested at different points in the competition, as reproduced in Table 2 [14]. Area of Excellence 1 (AoE 1) means that a CRS has to discover the function of the binaries they are given to analyse. Next, the ability to perform patching (AoE 2) is demonstrated by fixing flaws

Table 2: Evaluated Areas of Excellence

	Areas of Excellence	Tested at CQE?	Tested at CFE?
1	Autonomous Analysis	Yes	Yes
2	Autonomous Patching	Yes	Yes
3	Autonomous Vulnerability Scanning	Yes	Yes
4	Autonomous Service Resiliency	Yes	Yes
5	Autonomous Network Defense	No	Yes

in these same binaries. The third AoE (Vulnerability Scanning) is shown through the construction of input that proves vulnerabilities in binaries. Service Resiliency (AoE 4) refers to the ability to keep a binary running and available. Ultimately, the CRSs had to be capable of protecting a networked server (AoE 5).

The CQE, held in 2014, evaluated preliminary versions of the autonomous systems, and therefore only measured the first four AoEs. DARPA gave the competitors more than a hundred challenge binaries (CB), that contained bugs. For each CB, the competitors had to submit a Proof of Vulnerability (PoV), an input that demonstrated that the binary could be compromised. They also had to submit a replacement binary that patched this vulnerability. For each submitted replacement CB, a score was calculated according to the formula:

$$\text{CB Score} = \text{Availability Score} \times \text{Security Score} \times \text{Evaluation Score}$$

Without going into too much detail, the *Availability Score* (measuring AoE 4) was calculated by comparing the Replacement CB with a reference patched CB generated by DARPA, based on measures such as file size and memory usage. The *Security Score* (measuring AoE 2) was a measure of the proof of vulnerabilities (PoV) submitted to motivate the patches found in the replacement binary. Finally, the *Evaluation Score* (measuring AoE 3) evaluated each PoV submitted with a replacement CB, regardless of whether the vulnerability was patched or not. Notice that AoE 1 was not explicitly measured, but that the ability to find flaws and patch them was seen as enough evidence that a CRS was able to analyze binaries. So the CQE purely tested

the ability of these autonomous systems to analyze binaries, without testing their ability to protect a server.

The finals, in contrast, pitted the autonomous systems against each other in a Capture the Flag (CTF) tournament. Each autonomous system was given a server that contained the code for several flawed CBs. The goal was to "capture" opponents' flags by proving that one of their programs was vulnerable, and to prevent other teams from exploiting vulnerabilities in the programs they were defending. This tournament took place over several rounds, and each round, the CRSs were scored based on the same three criteria as the CQE, but measured differently. The criterions were:

1) Security (AoEs 1 &2): Patch vulnerabilities in the CBs to keep your flags safe.
2) Availability (AoE 4): Programs should continue running normally after they are patched
3) Evaluation (AoEs 1 & 3): Submit PoVs in opponents' software, which counted as captured flags.

This final thus tested all the Areas of Excellence outlined above, including the fifth, given that CRSs had to protect the binaries on their servers for about ten hours.

ii. Testing Infrastructure

To run this competition, DARPA had to develop a suite of testing tools and challenge binaries. Those who built the challenge binaries modeled CWEs, such as stack and heap overflows. DARPA also had to create a new computing environment: DECREE. Building this new environment allowed DARPA to ensure that 1) it was completely isolated from all networks, 2) the systems were encountering genuinely novel binaries, 3) the programs had determinate behaviour, and 4) that vulnerabilities would not affect any real world machines. This environment was simple, in that it had only seven system calls. Moreover, it severely restricted

inter-process communication. Processes did not share memory, and had to communicate via an intermediary. [15] All the testing tools, some challenge binaries and DECREE were made available to the general public in a github repository[4] to foster general interest and innovation.

IV. DEF CON 24: The CGC Finals & Mayhem vs Humans

i. CGC Finals

The CGC was about more than finding the best CRS, it was designed to engage the general public and pique their interest in using autonomous systems for network security. To that purpose, the finals of the CGC were held a DEF CON 24, and they were held as a spectator event. DARPA developed several visualizations [16], some of which are shown in Figure 1, to help those in the audience get a glimpse of the invisible battle happening within the servers. Visualization (a) showed the status of each CRS and the challenge binaries[5] they were protecting. The audience could also watch the path taken by input through code on 3D maps, as shown in (b). Moreover, DARPA created a visualization (c) to characterize the challenge binaries, allowing users to see how much a replacement binary deviated from its original, for instance. These visualizations allowed non-experts to get insight into the challenge, without necessarily having to understand the specifics of techniques like fuzzing or Symbolic Execution.

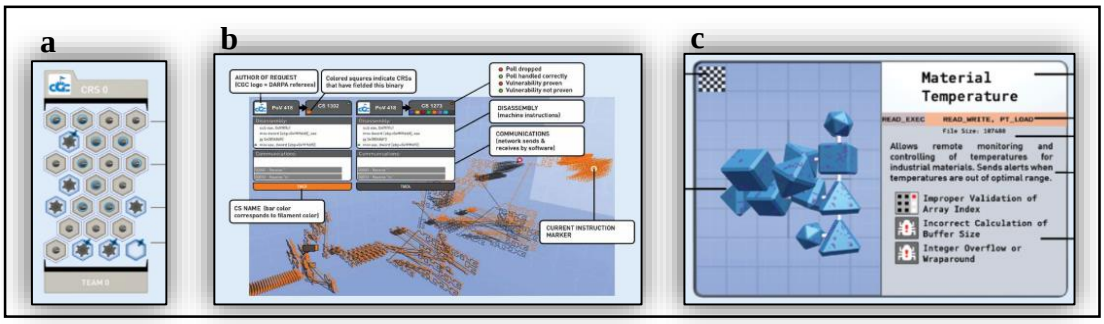

Figure 1: Visualizations of (a) CRSs' servers, (b) path of input through code and (c) challenge binary

[4] https://github.com/CyberGrandChallenge
[5] Renamed as "Challenge Sets" in the finals

Beyond visualizations, the hardware used to run the challenge was also eye-catching. Mayhem and the six other finalists were placed on towering servers, for all to see, on a stage flooded with colored lights. On this stage, for hours, the autonomous systems battled it out, without any help from their creators. (To guard against cheating, the stage was designed such that no wireless signals could reach the machines [17]). Finally, after 95 rounds, Mayhem was crowned the winner. The results of the CGC were reported in a huge number of publications, such as BBC News, Der Spiegel and the New York Times. So, DARPA's efforts to make the event noteworthy were not in vain, given that both the national and international press took notice.

ii. Mayhem vs Humans

Besides visibility, a reason for holding the CGC at DEF CON 24 was that the winner could then participate in the DEF CON's Capture the Flag event, against humans. However, the autonomous systems were only designed participate in CGC style competitions. Thus, the Legitimate Business Syndicate, the organizer of the CTF tournament, had to tailor the event to the CRS that would be playing along with the humans. The LBS ran its event in DECREE, the environment designed by DARPA for the CGC. It also adopted the scoring used by the CGC [18]. Yet, even though the competition was designed to accommodate the CRS, no one really expected Mayhem to win. Even Paul Walker, program manager of the CGC was quoted as saying: "I don't expect Mayhem to finish well. This competition is played by masters and this is their home turf. Any finish for the machine save last place would be shocking." [19] As predicted, Mayhem finished last, but it did put up a good fight. At some points during the competition, it even managed to surpass some human teams [18]. This small success illustrates

that while CRSs are not yet ready to replace humans, they might play a helpful role in aiding the humans who protect our networks.

V. Conclusion

We looked at the Cyber Grand Challenge from two perspectives: those who created the autonomous system, and those who put them to the test. We found, by examining the inner workings of Mayhem, that building autonomous systems requires thinking creatively about how to combine various algorithms and techniques. The key is to find complementary algorithms whose strengths mitigate the weaknesses of the others. Moreover, we saw that creating a testbed for such systems is far from trivial. DARPA produced new measures for these autonomous systems, and then had to develop a whole new environment, DECREE, to test them along these measures. Finally, we found that while these autonomous systems are able to detect vulnerabilities in programs and patch them, they are still outperformed by human experts.

Still, Mayhem would definitely outperform *me* in a CTF competition! Even in its current form, it is a system that could be extremely useful for protecting networks. However, like all tools, autonomous systems can be used both for good and bad. Any system that can find vulnerabilities and patch them, can also be designed to find vulnerabilities and exploit them. Thus, I foresee that, like most things in the security game, this is just the beginning of an arms race between the attackers and those protecting networks.

Works Cited

[1] Defense Advanced Research Pojects Agency (DARPA), "Darpa Announces Cyber Grand Challenge," 22 October 2013. [Online]. Available: http://www.darpa.mil/news-events/2013-10-22.

[2] DARPA, "Cyber Grand Challenge: CQE Scoring Document," 2014 7 July. [Online]. Available: https://cgc.darpa.mil/CQE_Scoring.pdf.

[3] DARPA, "Program: Cyber Grand Challenge," [Online]. Available: http://www.darpa.mil/program/cyber-grand-challenge.

[4] T. Rains, "How Vulnerabilities are Exploited: the root causes of Exploited Remote Code Execution CVEs," 24 June 2014. [Online]. Available: https://blogs.microsoft.com/microsoftsecure/2014/06/24/how-vulnerabilities-are-exploited-the-root-causes-of-exploited-remote-code-execution-cves/.

[5] "The Heartbleed Bug," Codenomicon, 29 April 2014. [Online]. Available: http://heartbleed.com/.

[6] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer and V. Paxson, "The Matter of Heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, 2014.

[7] A. Arora, R. Krishnan, R. Telang and Y. Yang, "An Empirical Analysis of Software Vendors' Patch Release Behavior: Impact of Vulnerability Disclosure," *Information Systems Research*, vol. 21, no. 1, pp. 115-132, 2010.

[8] T. August and T. Tunca, "Network Software Security and User Incentives," *Management Science*, vol. 52, no. 11, pp. 1703-1720, 2006.

[9] ForAllSecure, "Unleashing the Mayhem CRS," 9 February 2016. [Online]. Available: https://blog.forallsecure.com/2016/02/09/unleashing-mayhem/.

[10] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385-394, 1976.

[11] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo and D. Brumley, "Automatic Exploit Generation," *Communications of the ACM*, vol. 57, no. 7, pp. 74-84, February 2014.

[12] T. Avgerinos, A. Rebert and D. Brumley, "Methods and Systems for Automatically Testing Software". US Patent 0339217, 26 Nov 2015.

[13] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco and D. Brumley, "Optimzing Seed Selection for Fuzzing," in *Proceedings of the 23rd USENIX Security Symposium*, San DIego, 2014.

[14] DARPA, "Cyber Grand Challenge Rules," 18 November 2014. [Online]. Available: https://cgc.darpa.mil/CGC_Rules_18_Nov_14_Version_3.pdf.

[15] DARPA, "Machine vs Machine: Lessons from the first year of the Cyber Challenge," 12 August 2015. [Online]. Available: https://www.usenix.org/sites/default/files/conference/protected-files/sec15_slides_walker.pdf.

[16] DARPA, "Cyber Grand Challenge: Final Event Brochure," 2016 4 August. [Online]. Available: https://s3.amazonaws.com/cgcdist/cfe/cgc-final_event-cfe-brochure.pdf.

[17] DARPA, "Cyber Grand Challenge: CFE Event Plan," 2016 26 July . [Online]. Available: https://s3.amazonaws.com/cgcdist/cfe/darpa-cgc-cfe-event_plan.pdf.

[18] L. B. Syndicate. [Online]. Available: https://blog.legitbs.net/.

[19] DARPA, "DARPA Celebrates Cyber Grand Challenge Winnders," 5 August 2016. [Online]. Available: http://www.darpa.mil/news-events/2016-08-05a.

[20] S. K. Cha, T. Avgerinos, A. Rebert and D. Brumley, "Unleashing Mayhem on Binary Code," *2012 IEEE Symposium on Security and Privacy*, pp. 380-394, 2012.