AUTOMATED EXPLOIT GENERATION OF BINARY TARGETS BY LEVERAGING A CUSTOM FUZZING AND DEBUGGING FRAMEWORK

Samuel Cappella, Mike Jochen
East Stroudsburg University of Pennsylvania
sjc7437@live.esu.edu, mjochen@esu.edu

ABSTRACT
Software bugs are plentiful and costly. Due to the sheer quantity of bugs, prioritizing bug fixes can be difficult. In respect to information security, bugs that can be exploited should be of the highest priority in software maintenance. Producing a working exploit for a discovered vulnerability creates a sense of urgency to correct software defects in a timely manner. In this paper, we propose an automated solution to generate exploits for binary targets by leveraging a custom fuzzing and debugging framework. Our results have been successful and promising in the early stages of testing. To the best of our knowledge, we have developed the first end-to-end fuzzing based exploit generation system for binary targets.

KEY WORDS
Exploit, automated, debugger, fuzzing, binary, security

1. Introduction
Software bugs (defects or faults in software) are very costly to the economy. One report from the National Institute of Standards and Technology (NIST) placed the cost of software defects to the U.S. economy to be $59.5 billion annually in 2002 [1]. Software bugs take time and effort to discover. From a business or organization standpoint, there often is a cost/benefit analysis associated with the process of bug hunting. A business or organization may find it too costly to attempt to find and fix potential bugs. In addition to appropriating funds to conduct testing in order to identify software bugs, there may be too many found bugs to address at once, and the bugs must be fixed on a priority basis. From a security standpoint; bugs that can lead to security vulnerabilities exploitable by attackers should be of highest precedence. These defects, if left unchecked, could lead to active exploits that enable successful cyber attacks and cyber crime to occur, resulting in direct costs to a business or organization. One report places the average cost of cyber crime for a business to be $8.3 million annually [2].

In this paper, we present a prototype system to automatically detect software defects in a binary target that could lead to a vulnerability and possible exploit. Upon detecting a vulnerability, the system will automatically generate an exploit to demonstrate the existence of the vulnerability. The automatic binary exploit generating system is verified by a functional utility that demonstrates the completeness of our solution. Our utility provides an end-to-end solution to develop a working exploit for a specific binary target, given nothing more than the binary itself.

By generating working exploits for vulnerable bugs, we provide a solution to prioritizing bug fixes, based on demonstrable ill effects. By allowing our utility to evaluate binary targets, organizations can perform security and risk assessments of commercial off the shelf (COTS) software, even without the ability to access the target program’s source code. Without the need for program source code, our solution becomes applicable to a much wider range of targets.

For the purposes of this paper, we define an exploitable bug as a vulnerability that can lead to a control flow hijacking exploit such as arbitrary code execution. In addition, an exploit is a concrete example of a vulnerable bug being leveraged to alter the execution flow of a program. Our utility focuses on buffer-overflow exploit generation after detecting the ability to control the target binary’s Instruction Pointer (IP).

At a high level, our utility combines the process monitoring abilities of a debugger, the data generation and testing of a fuzzer, and implements a novel exploit generation module. Target binary programs are passed as command line arguments to our utility. Our system then executes the target program while monitoring changes in program state. Test cases are generated by our system to craft target program input in an attempt to identify program faults that could lead to vulnerabilities. Our utility starts the target binary and monitors the changes in the process state as the target binary executes based on various input test case generations.

The goal of our utility was to combine the functionality and usefulness of various tools used in common bug finding and exploit generation processes. We first needed a fuzzing tool that would simulate potentially malicious input data. In tandem with the fuzzer, a debugger was required to watch for harmful changes in process state.
The final piece is a module that analyzes output from the fuzzer and debugger when a vulnerability is found in order to generate a working exploit.

A fuzzer is a utility designed to find vulnerabilities in software by brute forcing the vulnerability finding process [3]. The basic function of a fuzzer is to generate test case data that can be used on an evaluation target. In our case, our evaluation target is the target binary we are examining for vulnerabilities. The goal of the fuzzer is to generate a set of input data to test against the target. Often, fuzzers will generate a series of test cases that cause the target binary to crash. An effective fuzzer will log all test cases leading up to the crash of the target so a reviewer can go back through and start to hone in on the specific problem.

Fuzzers are effective at finding potential vulnerabilities. Vulnerabilities detected by a fuzzer usually manifest in the form of a program crash [3]. Following the crash of a target binary, a researcher will examine the input data generated by the fuzzer on the previous run. In order to determine why the target crashed, the binary will be executed again using the data generated by the fuzzer, but with a debugger attached. With the debugger attached to the target process (the binary being examined), the researcher can step through each instruction and watch how the process state changes. These changes in process state can help the researcher discover why the process crashed. A specific state change a researcher may be looking for is a segmentation fault caused by the target process trying to execute an instruction at an invalid address. In fact, our utility currently monitors for this type of process state change.

When a running process iterates through a series of instructions, the IP is responsible for keeping track of the next instruction to execute. We can control the flow of execution if we can control what gets loaded into the IP [4]. Control flow hijacking exploits alter the flow of execution commonly by controlling IP [3]. Our solution utilizes the debugger to watch the IP (specifically the EIP x86 machine register) until it contains data generated by the fuzzing module. If the IP changes to a controlled data value, we move on to the next stage and generate the exploit, a buffer-overflow.

One of the most common control flow hijacking exploits is a buffer-overflow exploit [5]. A buffer-overflow occurs when a buffer, a piece of contiguous memory, gets filled with more data than it was meant to contain [5]. The data essentially over-flows the buffer (hence the name). When this overflow occurs, data residing next to the buffer in memory can become corrupt. Due to the organization of the stack on 32-bit x86 systems, the machine register responsible for maintaining the value of the IP, the extended instruction pointer (EIP), can be overwritten. An attacker is able to control this over-write and change the value of EIP so program execution flow is altered.

Our solution takes all the steps necessary to fuzz the target program with generated test cases, watch for the IP to become corrupt, and develop an exploit that can successfully alter program control flow.

2. Related Work

Work in the area of automating the software testing process has been conducted for quite some time [6] [7]. However, the area of automated exploit generation has been a relatively new area of research. The first known work presenting a working system for automatically generating exploits comes from Carnegie Melon University that demonstrated the ability to automatically generate exploits based on the differences in binaries and patches [8]. In this early work, the differences in code between the original target binary and the update patch are used to deduce the vulnerability that is addressed by the patch. More recent and notable research presentations have been [9] [10] [11]. The AEG system is able to identify bugs in software by searching execution paths. These bugs are used to generate working exploits against the target software. The AEG system is able to generate new exploits in a very short time.

Carnegie Mellon University advanced their research of AEG to develop MAYHEM. MAYHEM presents the first end-to-end exploit generation system for binary targets. MAYHEM is unique in that it leverages a separate tool (also developed by Carnegie Mellon University) called Binary Analysis Platform (BAP) [12]. With BAP, MAYHEM raises the executable assembly code in a binary to an intermediate representation language. This intermediate representation is then fed through a symbolic execution engine. By symbolically executing the intermediate representation language, constraints are generated to describe error states. Constraints generated by the symbolic execution engine are symbolic in nature and must be solved for. A satisfiability modulo theory (SMT) solver is used to generate concrete values from symbolic constraints. The generation of concrete values provides the required input to drive the target binary into an exploitable error state.

In order to symbolically execute all the intermediate representation language code, MAYHEM makes use of a virtualization layer to simulate operating system calls. The virtualized system calls can act as network interfaces, file interfaces, and other standard forms of input. The advantage to this approach is the ability to fine-tune the data generation techniques from each interface. The downside is the requirement to re-implement every system call as a virtual interface. As such, MAYHEM does not have support for every system call and will not work on every binary.

To contrast the previously mentioned works, our solution appears to offer the best method for controlling the execution state explosion. Due to the approach taken by
other researchers, the memory use of their tools is extremely high. As such, scaling to try and evaluate large programs is extremely difficult and may even be impossible in some cases. Our approach offers a solution to this problem by sacrificing total run-time. In addition, since there is no virtualization layer to implement, every binary is a potential target.

Del Grosso, et al. present a method to utilize genetic algorithms to generate test case input without the need for human intervention during the test case selection/generation process [13]. The authors leverage both static and dynamic information to detect buffer overflows in software under test. This work follows a white box testing model, where access to the software source code is required. Our work follows a black box testing model, where access to system source code is not assumed.

3. Project Overview

In this section we describe the overall utility architecture and expand upon the ideas presented in the Introduction. Each component of the utility will be described in detail and how they solve specific challenges pertaining to automatic exploit generation of binary targets.

Up to this point, the primary focus of our utility has been to focus on command line argument attack vectors. Currently, we do not have support for providing a fuzzing interface from other mock sources of input such as network sockets, file IO, or interactive user input. It is also important to note that our utility was designed to run on a modern x32 Linux-3.2 system.

3.1 Fuzzing Module:
The goal of the fuzzer is to generate a meaningful set of test data from the set of all possible inputs [3]. Human testers can sit and attempt to test for every possible input but that would be impossible. The set of all possible inputs is infinite [3]. In addition, a human tester is slow. To accommodate these limitations, fuzzers offer a means to automate the process of data generation for test cases and running the test.

The design of our fuzzer focuses on constructing character arrays to be used as a command line argument when the target binary is invoked. The challenge we face is what constitutes a meaningful character array. The set of all possible character arrays that exist using ASCII characters is impossibly large. Let’s consider a simpler data type. For instance, fuzzing a data field that is of type integer has $4,294,967,296$ possible values. A fuzzer would have to execute a test for every possible integer value to exhaust all possibilities. Even if we could test $10,000$ different integer values every second, it would still take about $429,000$ seconds to test every integer value, and that is for just one variable. As the number of variables increases, the run-time and data input possibilities increase exponentially [3].

When designing our fuzzing module, we needed to implement a solution that would generate meaningful data. Since our goal was to generate data that would lead to the compromise of the IP, we wanted to easily detect when EIP reflected a changed IP. A common pattern in vulnerability discovery is to fuzz target programs with strings consisting of repeating instances of the letter ‘A’ and gradually append additional ‘A’s to the string after each iteration [3]. When fuzzing with strings of ‘A’s and a successful corruption of IP occurs, the target encounters a segmentation fault by trying to execute an instruction at the memory address $0x41414141$. This memory address is significant because the hexadecimal ASCII value of the character ‘A’ is $0x41$. Essentially, we have written over the IP with our input generated by the fuzzer. The fuzzing module does not determine when the IP has been over-written. The only role of the fuzzer is to generate the data and log the test case. The debugging module is responsible for determining the process state change via IP over-write.

3.2 Debugging Module:
A debugger, such as the GNU Project Debugger (GDB), is a tool used to see what is going on ‘inside’ a program while it executes or to examine what happened when another program crashes [14]. For our purposes, we needed a way to inspect a process the moment the IP gets over-written with data generated by the fuzzer module. Our initial solution was to use GDB as our debugger of choice due to its prolific reputation in vulnerability discovery and exploit development [4] [5] [15]. The standard method of interfacing with GDB is to invoke GDB with a variety of command line arguments to set up an interactive debugging interpreter. This interpreter session exposes all the features of GDB to the researcher doing the debugging. This, however, was not what we needed. In order to automate the debugging process, our solution needed a mechanism to interface with GDB programmatically.

One feature of GDB is the ability to invoke GDB with a variety of interfaces. One in particular that seemed promising was the GDB/MI interface. The GDB/MI interface is a line based machine oriented text interface to GDB [16]. The GDB/MI interface is designed to specifically work with a machine like process through command and response communication. The challenge then became to develop an Application Programming Interface (API) to communicate directly from our fuzzer module to GDB through the MI. Unfortunately, MI was not mature enough to meet our needs. The landing page of the GDB/MI project even states, “[n]ote that GDB/MI is still under construction, so some of the features described below are incomplete and subject to change (sic) [16].” As a result, while attempting to develop our API, progress
was slow and complex. To circumvent this problem, we developed our own debugger.

Our debugger needed to meet three requirements: A) the debugger must be able to execute the target program instruction by instruction, B) monitor the register values and watch for EIP to reflect the data generated by the fuzzer, and C) determine where the stack pointer is at the time of the crash and the contents of the stack. To accomplish these goals we utilized a couple of system level tools that GDB also uses. The most predominant was the ptrace() system call and the other was the sys/user.h header file written specifically for GDB.

Execution tracing is a technique that allows a process to monitor the execution of another process [17]. In Linux, the ability to do execution tracing is made possible by the ptrace() system call. With ptrace(), we are able to accomplish the goal of being able to execute a target program instruction by instruction. The way ptrace() works is by forking a child process. The child process uses a PTTRACE_TRACEME request to the parent. Once the trace request is established, the parent process has total execution trace control over the child. In our implementation, we then use the child process to start executing the target program. By result, our parent process now has total control over our target program. Once the target program has been started with the new iteration of the fuzz string, the utility executes the target program instruction by instruction and during every step, monitors the instruction pointer.

As previously mentioned, the sys/user.h header file was designed specifically for GDB. The purpose of the user.h header file is to provide a data structure to store the register values of a program being traced. In addition to reading register values, we required the debugger to be able to read the stack when the traced program crashes.

The debugger implemented in our utility has the ability to trace the execution of a target process, monitor every change to the registers utilized by the target process, and dump the contents of the stack. More importantly, our debugger integrates with the fuzzing module. Before every new execution of running the target process, the debugger gets a new command line argument from the fuzzer and supplies it to the target program. We previously described how common vulnerability discovery will fuzz a target program and watch for the IP to turn to 0x41414141. That task is assigned to the debugger. After every instruction execution, the debugger checks the value of the EIP register. If EIP is equal to 0x41414141, the debugger stops executing the target program and dumps the current stack contents.

The dumping of the stack is required to verify the fuzz data has successfully overwritten the IP. If the IP was successfully overwritten by the fuzz data, there should be a contiguous block in memory confirming this to be true.

If the corruption of the IP turns out to be true, the exploit generation module takes as input the data that caused the corrupt IP and the value of the stack pointer (ESP) when the target program crashed.

3.3 Exploit Generation Module:
Up to this point we have described the process of identifying vulnerabilities. The next step is to generate a working exploit for the vulnerability. The process of generating a working exploit is relatively straightforward but it is necessary to understand how create a buffer-flow exploit first.

With a buffer-overflow, the goal is to redirect program execution in such a way that the target program executes code of the attackers choosing. To do this, there must be an address to direct control flow to. At a high level, an attacker will place the address to jump to at the end of the input string, the instructions to execute, and then NOP (no instruction) padding in front of the malicious instructions. A more detailed explanation is provided in [4][5][15].

Once the target program crashes and the IP has been corrupted, the debugger and fuzzer pass data along to the exploit generation module. The exploit generation module requires the fuzz string that caused the crash and the ESP value at the time of the crash.

The first step in generating the exploit is to calculate the size of the attack buffer. The size of the buffer is easily calculated by counting the length of the fuzz string. Due to the implementation of the fuzzer, the debugger is able to detect the minimum length string needed to corrupt the IP. Thus, the length of our attack buffer is the same length of the fuzz string that corrupted the IP. The next step is to calculate the memory address to redirect control flow to.

The ESP value passed to the exploit generation module points to the where the top of the stack exists in memory at the time of the crash. Due to the nature of how the stack is laid out, the target address to redirect control flow to can be calculated by subtracting the size of the attack buffer from the value at ESP once the program crashed plus an offset proportional to the buffer size. Once the address is calculated, it is placed at the end of the attack buffer in reverse order (to accommodate endianess).

Once the new IP address has been calculated, the code of the attackers choosing (currently our utility uses generic shell code) is prepended to the buffer in front of the new IP address. The final step is to fill out the buffer to increase the success rate. The NOP instruction, \x90, is used to pad all extra space in the attack buffer. After the exploit string has been created, the exploit generation module verifies the exploit.

To verify the exploit, the target program is run once more in process trace mode. The debugger watches for EIP to become corrupt with the target redirect address calculated.
and ensures that when EIP is corrupt with the redirect address, the extractions to execute at that location are the padded NOPs. If the value at the address is indeed the NOP instructions, the verifier will scan through the memory address for the length of the buffer and ensure the entire exploit string made it into memory correctly.

4. Experimentation and Evaluation

Our utility was evaluated on a 32-bit Ubuntu 12.04 virtual machine system image with a 2.33Ghz Intel Xeon and 2Gb RAM. Due to the direct interaction with the Linux kernel, the utility is written in roughly 550 lines of C.

As a proof of concept, it is important to note that our utility does not circumvent modern operating system buffer-overflow protections such as ASLR or DEP. Although we do have plans to integrate those features in the future, the current goal was to first develop a working proof of concept. As an additional note, these limitations also apply to [10] and [11].

To evaluate our utility, we used a collection of programs, all with a strcpy() buffer overflow vulnerability that was reachable from a command line argument. The programs varied in number of lines of code, the number of variables declared, and the size of buffer, among other criteria. For our test cases, our utility proved to be 100% effective. Every binary we tested against, our utility found the vulnerability and was able to generate a working exploit.

In regards to performance, total memory consumption is only slightly greater than that of the target binary while running. Runtime, however, is typically between five and ten minutes for programs with buffer size of 120 or less. Due to the nature of repeatedly running and fuzzing the target, time grows linearly as the size of the buffer increases.

We are currently in the preliminary states of experimentation and evaluation. We have only recently completed the prototype implementation of our system and have started empirical evaluation and testing. While initial results are very promising, we look forward to presenting more rigorous tests results in the near future.

5. Future Work

There are many milestones on the map in the area of future research directions and extensions to our work. The first goal is to perform more testing on a variety of programs to evaluate a more precise time ratio in terms of program size and buffer size. We also plan on collecting better memory use metrics through the use of tools such as Valgrind [18].

We also have plans to expand on the utility as a more flexible tool. A major goal is to try and improve run-time performance by experimenting with a distributed version of the utility, add the option for custom attack code or provide a list of options, and expand beyond the limitation of just command line based attack vectors.

6. Conclusion

We presented a utility for automatically generating working exploits for binary targets by leveraging a custom fuzzing and debugging framework. Or solution is, to the best of our knowledge, the first of its kind to provide a complete end-to-end solution for automated exploit generation without access to the target software source code. Our utility also offers a solution to evaluating larger target programs in order to scale vulnerability detection and exploitation. Preliminary results show that our prototype system achieves the goal of identifying software faults and generating a working exploit against the newly found vulnerability.

8. Acknowledgements

We would like to thank the PACISE anonymous reviewers for their helpful comments and feedback. This research was funded in part by a research grant provided by SPAWAR. The content of this paper does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

References:


