Automatic Buffer Overflow Exploit Generation from File and Socket I/O

Anthony Gavazzi
Northeastern University
gavazzi.a@northeastern.edu

***Abstract*—Control-flow hijacking vulnerabilities, such as those induced by buffer overflows, are among the most security critical software vulnerabilities, as they can lead to arbitrary code execution and, potentially, complete system compromise. Many tools have been developed over the past several decades to help developers find and eliminate such vulnerabilities. One open source tool that can do this is Zeratool, which leverages the symbolic execution and binary analysis tool angr to find vulnerabilities in binaries and generate exploits for them.**

However, Zeratool only supports exploit generation when the exploit is triggerable via standard input, making it ineffective at finding vulnerabilties exploitable via socket or file I/O. Thus, this paper presents an extended version of Zeratool that tracks the symbolic contents of all open file descriptors to detect buffer overflow vulnerabilities exploitable by socket and file I/O. The extended tool was evaluated on two simple executables and can consistently find both vulnerabilities and generate exploits for them, even when the stack is not executable. However, the exploits do not work as-is due to angr's inability to determine the actual address the overflowed buffer will take at runtime.

I. Introduction

Many enterprise-level applications are written in C or C++ due to the flexibility and high runtime performance the languages offer. However, this performance comes at a cost, as neither language provides built-in security mechanisms such as buffer boundary checking. The result of this is that C and C++ code may contain buffer overflow vulnerabilities in which a write to a buffer results in writing data beyond the bounds of the buffer. By carefully crafting program inputs that overflow such buffers, attackers can overwrite variables related to the program's control flow and achieve arbitrary code execution, which, depending on the program being exploited, can lead to anything from denial-of-service to complete system compromise.

Given the security-critical nature of these vulnerabilities, numerous tools have been developed over the last several decades to find such vulnerabilities so that they can be patched. These tools range from simple linters to extensible symbolic execution systems, and may be open-source or proprietary. One such open source tool is Zeratool [1], which leverages the binary analysis framework angr [2, 3, 4] to symbolically explore execution paths of binaries to find control-flow vulnerabilities and generate working exploits.

However, the tool has several major limitations. First, the tool was designed to generate exploits for capture-the-flag (CTF) challenges, where one exploits binaries on a remote server by sending exploit strings over standard input. This means that Zeratool can only detect vulnerabilities that are exploitable via standard input, making the tool effectively useless at finding many vulnerabilities in the real world (e.g., those that are exploitable by reading malicious data from a file or socket). Additionally, Zeratool is outdated, having been written for Python 2 and angr 7.8.2.21. The tool is also poorly written, with numerous redundant or unnecessary checks that slow down the analysis.

This paper presents an extended version of Zeratool, written for Python 3 and angr 9.0.6421, that can detect and generate exploits for control-flow hijacking vulnerabilities (such as buffer overflows) triggerable by reading malicious data from files or sockets. The extended tool takes an ELF executable as input and symbolically executes it, analyzing the memory addresses controlled by all open file descriptors to see if these addresses could store shellcode and whether the program counter could point to that shellcode. If such conditions are found to be satisfiable, Zeratool identifies the file descriptor that was read to exploit the vulnerability and dumps its contents to a file for use in exploiting the actual vulnerability.

Two exploitable binaries were created to evaluate the tool. The first is based on a real vulnerability in the Visual Boy Advance game emulator, which had a buffer overflow vulnerability exploitable by reading from a malicious file. The other is a simple server that contains a buffer overflow vulnerability. Zeratool can detect both vulnerabilities and generate exploits for them in human time. However, the exploits strings do not work as-is because angr cannot determine the actual address the overflowed buffer will take at runtime.

II. Related Work

Numerous tools have been developed over the past several decades to detect buffer overflow vulnerabilities. This section will discuss five such tools.

Haugh and Bishop developed STOBO [5], which detects buffer overflows by dynamically tracking the sizes of memory buffers and reporting vulnerabilities if the inputs to any of nine potentially vulnerable C standard library functions could overflow the buffer. As this is a dynamic technique, it is more effective than linting and can identify when unsafe functions such as strcpy are used safely. However, it can only detect vulnerabilities induced by those nine functions. If a user or third-party supplies their own implementation of the same functionality, STOBO cannot find vulnerabilities that arise from using it. Additionally, overflows from bugs such as uninitialized variables cannot be detected.

A more sophisticated tool is the automatic exploit generation (AEG) tool developed by Avgerinos et al [6]. This was the first work to present control-flow hijacking exploits as a formal verification problem that can be detected with symbolic execution. Given the source code of a program, AEG uses preconditioned symbolic execution to find exploitable vulnerabilities and then uses dynamic execution to generate working exploits. AEG is highly accurate and does not generate false positives, but only detects stack-based buffer overflows and cannot handle more complex vulnerabilities. Also, because it requires source code, it cannot be applied to binaries for which the source code has been lost.

AEG's successor, Mayhem [7], introduced hybrid symbolic execution to find bugs in a similar method to AEG, but with improved memory management and with mechanisms to avoid repeating repeated instruction execution. Mayhem operates on binaries with no source code, so it can be applied to binaries for which the source code has been lost, but, as a result, it is slightly slower than AEG. Another limitation of the tool is that it does not model all system calls and library functions, so it cannot handle large and complex binaries.

Ding et al. [8] proposed a technique for pattern-based symbolic execution that involves trimming execution paths that do not match certain patterns of syntax usage, element accessing, and bulk data moving. The approach operates on control flow graphs generated from source code, so, like AEG, it cannot be applied to raw binaries. The approach works well on large code bases, but misses many vulnerabilities for the same reasons as STOBO, and reports many false positives because it cannot identify effective input sanitization.

One final tool to discuss is Sys [9], an extensible static and symbolic execution framework to find bugs in extremely large code bases of LLVM bytecode. Sys works in two phases. In the first phase, user-written checkers are applied statically to the code to flag several potentially-vulnerable error sites. Then, each of these sites are symbolically executed with user-written symbolic checkers to verify whether a vulnerability truly exists. The approach scales well to large code bases and can reason about overflows arising from complex conditions such as integer overflows, but is only as accurate as the user-written checkers. Additionally, since the flagged sites are not symbolically executed from the program entry point, the tool lacks complete precision.

III. Approach

This section provides an overview of the techniques and tools Zeratool uses to find exploitable vulnerabilities. We first discuss angr and its symbolic execution capabilities, then describe the general workflow Zeratool goes through to generate exploits, as well as the challenges that were overcome to make Zeratool more accurate and flexible.

A. angr and Symbolic Execution

Zeratool finds exploitable vulnerabilities primarily by leveraging angr's symbolic execution capabilities. Starting at the entry point of a binary, angr conceptually executes the program one block of machine code at a time, similarly to how an interpreter would. angr maintains a "simulation state" that reflects the current state of the program being executed. Any inputs to the program (i.e., values under user control) are assigned symbolic, rather than concrete, values. When an operation occurs that uses symbolic values, the result is another symbolic value in terms of the input symbolic values. For example, if one were to execute y = x + 5 where the value of x is symbolic, the value of y would be "x + 5" rather than a concrete integer. Executing y * 2 would result in the expression "2 * (x + 5)" and so on.

For a given simulation state, angr tracks various constraints on the possible values symbolic variables can take. Given a variable x with a symbolic value, if the program executes the statement assert(x != 0), then the constraint "x != 0" is added to the simulation state. When angr reaches a conditional statement where the outcome depends on a symbolic variable, then it forks the current simulation state, generating one new state for each possible branch of the conditional statement and applying corresponding constraints to each state. For example, if the code "if (x > 1) y = x - 1;" is executed and x has a symbolic value, then two new states are created, one with the constraint "x > 1" and the other with "x <= 1."

However, angr may not always have to fork at a conditional statement if the current constraints can guarantee that some branches are impossible. For example, if the code "assert(x != 0); if (x == 0) y = x - 1;" is executed, then the first assert statement will guarantee that x != 0, so angr will reason that the body of the if statement will never be executed, and thus will not fork the state. angr reasons about these conditions by passing all its current constraints to the Z3 Theorem Prover, which will return SAT if the conditions can be satisfied or UNSAT otherwise. angr provides an interface for a developer to query the Z3 solver directly to check whether various constraints are possible in a given simulation state.

Part of angr's utility is its ability to manage and execute multiple simulation states automatically. Simulation states are organized into stashes. Which stash a state belongs to depends on various conditions specified either by angr or by the developer. One stash managed automatically by angr is called "unconstrained," and contains all states where the instruction pointer is at least partially symbolic (i.e., controlled by user input). When states appear in this stash, it is likely that angr has found an exploitable state.

B. How Zeratool Finds Vulnerabilities

To find vulnerabilities, Zeratool first simply loads a binary using angr and steps through it until a state appears in the unconstrained stash. Once one has been found, it obtains a list of all memory addresses with contents under user control. It does this using another powerful feature of angr: its reverse memory name map. While executing a binary, angr can map all memory addresses back to the variables or file descriptors that wrote to them. Zeratool compiles all memory addresses that were written to with data read from standard input.

Then, for each of these addresses, Zeratool queries the Z3 solver to see whether 1) the instruction pointer could equal that address, and 2) whether a hard-coded /bin/sh shellcode could be loaded in memory at that address. If both conditions are satisfiable, they are added to the current state as constraints. Zeratool then obtains a dump of all data read from standard input and dumps that to a file.

This approach, however, only works if the binary has an executable stack. A common approach to get around non-executable stack is to use a rop-chain, which Zeratool is capable of generating. Before executing the binary, Zeratool inspects its ELF headers to check for non-executable stack. If the binary has this protection, then Zeratool uses the ropper library to generate a /bin/sh ropchain, and tries to load it into memory instead of the /bin/sh shellcode.

C. Extending Zeratool

One of Zeratool's major limitations is that it can only find vulnerabilities exploitable by reading from standard input. Extending Zeratool to find vulnerabilities exploitable by reading malicious data from files or sockets can be achieved by inspecting memory addresses controlled by any open file descriptors, not just file descriptor zero. The rest of the analysis proceeds as normal, except that if the exploit file descriptor corresponds to an actual named file that was read, then the name of that file is obtained and a file of that name is written to. If the exploit file descriptor corresponds to a socket, then one file is written for each read that is performed on that socket.

An additional limitation of Zeratool is that it does not make any attempt to control the maximum read size during symbolic execution. This is a critical aspect for both the accuracy and performance of the tool. Consider, for example, a program where a static buffer of size 256 is overflowed to achieve arbitrary code execution. By default, angr will only read 256 bytes of data from a file at a time. Thus, without increasing this value, angr may not read enough data to overflow the buffer and find the vulnerability. One could manually set the maximum read size to some large number like 65536 to account for all common buffer sizes, but this may unnecessarily cripple the tool's performance, as the Z3 solver will have to reason about 65536 distinct memory locations every time it is queried.

The developers of AEG and Mayhem found that the maximum read size should be at least ten percent larger than the largest allocated buffer in the program. This is easy to extract from source code or LLVM bytecode, but angr works on binaries alone. To approximate the largest buffer size, we make use of the fact that at the beginning of functions in assembly, the stack pointer is decreased to make room for all local variables in the function. We thus generate a control-flow graph of the program using angr and traverse it, looking for any instructions where the stack pointer is decremented by some constant value. We find the largest such constant value in the program and set the maximum read size to be ten percent larger than it. The value we find accounts for the size of all variables in a function, not just buffers, so this approach tends to overestimate the maximum read size. However, this should only pose issues in cases where functions allocate an enormous amount of space for variables after the overflowed buffer.

Zeratool was implemented in Python version 3.8.5 and with angr version 9.0.6421. The source code can be found at https://github.com/Gavazzi1/Zeratool. Within the repository is a Dockerfile that specifies the versions of various additional dependencies.

IV. Experiments and Results

To evaluate the extended Zeratool, two simple exploitable binaries were prepared. All development was performed on an Ubuntu 20.04.1 virtual machine with address space layout randomization (ASLR) turned off.

A. Visual Boy Advance Buffer Overflow

The first test binary was based on a vulnerable piece of code from version 1.8.0 of the Visual Boy Advance Emulator [10]. The vulnerable part of the code is shown in Fig. 1. The vulnerability is a buffer overflow. If the file being read makes len greater than 1024, then the buffer will be overflowed allowing an attacker to overwrite the return address and execute shellcode. The code was compiled with gcc version 9.3.0 via the following command "gcc -m32 -no-pie -fno-stack-protector -z execstack -o vba vba.c"

```
void go() {
    char buffer[1024];

    FILE *f = fopen("file.txt", "r");
    if (f == NULL)
        return 1;

    int games = 0;
    int len = 0;
    fseek(f, 0, SEEK_SET);
    fread(&games, 1, 4, f);
    while (games > 0) {
        fread(&len, 1, 4, f);
        size_t val = len;
        fread(buffer, 1, len, f);
        buffer[len] = 0;
        --games;
    }
}
```

Fig. 1. Source code for the file-based buffer overflow vulnerability

First, to measure the effectiveness of the tool, Zeratool was run on the vba executable five times. The average time of all five runs was 20.5 seconds with a standard deviation of 0.388 seconds. Each time, it successfully detected a vulnerability and created a file called file.txt. This alone demonstrates that Zeratool associated the vulnerability with the correct file. Next, we analyzed the contents of file.txt using the hexdump tool. The output of this is shown below.

```
00000000  01 00 00 00 a3 04 01 80  |........|
00000008  31 c0 50 68 2f 2f 73 68  |1.Ph//sh|
00000010  68 2f 62 69 6e 89 e3 50  |h/bin..P|
00000018  53 89 e1 b0 0b cd 80 00  |S.......|
00000020  00 00 00 00 00 00 00 00  |........|
*
00000410  00 00 5c fb 5c fb 7f fb  |..\.\...|
00000418  5c fb fe 7f 00 00 00 00  |\.......|
00000420  00 00 00 00 00 00 00 00  |........|
00000428
```

The first data read from file.txt is an integer representing the number of loop iterations to perform. Converting the first four bytes of the hexdump from little endian gives a value of 0x1. In order to generate an exploit, the inner loop must be evaluated at least once, so Zeratool was able to generate a correct value.

Next, another integer is read from file.txt representing the number of bytes to read. Converting bytes four through eight of the hexdump from little endian gives a value of 0x800104a3. The variable len is a signed integer, so the value of len will be negative. However, len is passed to fread, which takes an unsigned integer for its third argument, so the number of bytes that will be read is 2147550371. This is an enormous amount of data, and is certainly more data than is in file.txt. This is inconsequential when it comes to reading the file itself, as fread will simply reach the end-of-file and return. However, when the execution reaches line 19, it will try to write a value of zero far out of bounds and likely trigger a segmentation fault. We reached out to the angr developers about this discrepancy, and they were not sure why this was happening.

Bytes nine through thirty-one are the shellcode. Since angr found that the binary had an executable stack and that the instruction pointer could point into the buffer, the shellcode was loaded into the buffer itself. The file then contains zeros for 1011 bytes. This is simply padding up to the point where the return address is overwritten. The six bytes 0x5cfb5cfb7ffb are various bytes from the jump address but occur before the actual jump address. These are likely artifacts generated when angr tries to concretize the symbolic contents of the file, and can be ignored. The bytes 0x5cfdfe7f comprise the jump address in little endian, which should overwrite the return address completely on the stack. The zeros following this address have no significance, and were generated by angr for reasons unknown and unimportant.

Because ASLR was turned off on the machine, stack addresses should always exist at the highest addresses. Since this program is a stack-based buffer overflow, one would expect the jump address to begin with 0xf. However, the jump address found by angr begins with 0x7. This is because of an inherent limitation of purely static symbolic execution. Because angr does not execute the binary dynamically, it has no idea what the actual address of the buffer will be at runtime, and simply tries to make a good guess at what its address might be. AEG faced the same issue, and generated concrete exploits by passing the output of the symbolic execution to a dynamic analysis step which would determine the real address of the buffer. The result of this is that Zeratool cannot generate working exploits when it has to jump to an address on the stack. However, it is effective as a vulnerability-finding tool, as a developer can simply obtain a jump address themselves and write it in the file over the existing jump address.

Running the executable such that it reads from the newly generated exploit file results in a segmentation fault. Analysis using gdb reveals that, as expected, the write on line 19 is the cause. To test whether angr generated sufficient padding to overwrite the jump address, bytes four through eight of file.txt were overwritten to be 0x20040000, which when converted from little endian gives 0x420, the length of file.txt after the first eight bytes. Running the executable again in gdb showed that the program jumped to 0x7ffefb5c, which is exactly the jump address found by angr.

B. *Simple Echo Server*

The second exploitable binary was a simple echo server written in C. The vulnerable code for this server is shown in Fig. 2. This too is a buffer overflow vulnerability. If a malicious client specifies a size greater than 32, then the buffer is overflowed and the return address may be overwritten. The server was compiled using the same gcc version and flags as the first executable.

```c
void go(int sockfd)
{
    char buf[32];
    size_t len;

    read(sockfd, &len, sizeof(len));
    read(sockfd, buf, len);

    write(sockfd, buf, len);
}
```

Fig. 2. Source code for the socket-based buffer overflow vulnerability

Zeratool was run on the server executable five times. The average time of all five runs was 8.436 seconds with a standard deviation of 0.184 seconds. Each time, it successfully detected a vulnerability and created two files: server-exploit.0 and server-exploit.1. Each file corresponds to one read from a socket. Since two reads are necessary to overflow the buffer, this behavior suggests that Zeratool found the correct vulnerable site. Next, we analyzed the contents of both files using the hexdump tool. The output of this is shown below.

```
server-exploit.0
00000000  39 00 00 00              |9...|
00000004

server-exploit.1
00000000  31 c0 50 68 2f 2f 73 68  |1.Ph//sh|
00000008  68 2f 62 69 6e 89 e3 50  |h/bin..P|
00000010  53 89 e1 b0 0b cd 80 00  |S.......|
00000018  00 00 00 00 00 00 00 00  |........|
00000020  00 00 00 00 00 00 00 00  |........|
00000028  00 00 00 00 f0 fe fe 7f  |........|
00000030  05 00 00 00 00 00 00 00  |........|
00000038  00                       |.|
00000039
```

The first packet sent to the program is simply 0x39. The fact that server-exploit.1 is 0x39 bytes long demonstrates that unlike the other exploitable binary, Zeratool generated a variable specifying an amount to read and then sent exactly that much data. The first 23 bytes of server-exploit.1 are the shellcode. They are followed by 21 bytes of padding, then the jump address, then artifacts generated by angr when concretizing the symbolic contents of the socket, which can be ignored.

Like before, the jump address begins with 0x7 rather than 0xf as would be expected for a stack-based buffer overflow, so Zeratool is unable to generate working exploits. Running the server and then connecting to it via a socket and sending the exact data in the server exploit files results in the server attempting to jump to address 0x7ffefef0 exactly as appears in the generated exploits. Just like in the file exploit, Zeratool is able to reason about how much padding is needed to overwrite the return address.

V. Conclusion and Future Work

This paper presented an extended version of Zeratool that can detect control-flow hijacking vulnerabilities, such as buffer overflows, where the malicious data that triggers the exploit is read from a file or socket. The entire Zeratool codebase was ported to Python 3 and adapted to work with angr 9. The tool was evaluated on two simple executables, one of which was based on an actual buffer overflow in the Visual Boy Advance emulator, and the other of which was a simple echo server written by the author.

For both executables, Zeratool was able to detect the vulnerability and compute the correct amount of padding needed from the start of the buffer to overwrite the return address on the stack, all in a reasonable timeframe. However, since angr uses purely static symbolic execution, it could not determine the actual address the buffer would take at runtime, so the /bin/sh shellcode was never executed when attempting to perform the exploit. Also, for the Visual Boy Advance overflow, angr generated an input that resulted in a write so far out of bounds that it triggered a segmentation fault before the exploit could occur.

Much work could be done to improve Zeratool. For one thing, incorporating a dynamic component to determine the runtime address of a buffer would allow the tool to generate working exploits. Zeratool also works by using brute force symbolic execution, which is known not to scale to large code bases due to the path explosion problem. Combining Zeratool with a static analysis component to reduce the amount of computation it performs would drastically improve the tool. Additionally, Zeratool only works for simple, stack-based buffer overflow vulnerabilities. Much work could be done to detect and exploit vulnerabilities related to operations on the heap.

VI. Contributions

All work towards the completion of this project was performed by Anthony Gavazzi.

References

[1] Gavazzi1/Zeratool [Computer software]. (2014). Retrieved from https://github.com/Gavazzi1/Zeratool
[2] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. (2016). SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis.
[3] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. (2016). Driller: Augmenting Fuzzing Through Selective Symbolic Execution.
[4] Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., and Vigna, G. (2015). Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.
[5] Haugh, E., and Bishop, M. (2003). Testing C Programs for Buffer Overflow Vulnerabilities.
[6] Avgerinos, T., Cha, S. K., Lim Tze Hao, B., and Brumley, D. (2011). AEG: Automatic Exploit Generation
[7] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. (2012). Unleashing Mayhem on Binary Code
[8] Ding, S., Tan, H. B. K., Liu, K., Mahinthan, C., and Zhang, H. (2012). Detection of Buffer Overflow Vulnerabilities in C/C++ with Pattern Based Limited Symbolic Evaluation.
[9] Brown, F., Stefan, D., and Engler, D. (2020). Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code.
[10] x3ro/VisualBoyAdvance [Computer software]. (2006). Retrieved from https://github.com/x3ro/VisualBoyAdvance/blob/master/src/win32/GSACodeSelect.cpp