Exploot: Vulnerability Detection Through Automated Exploit Generation

Sami El Feki - selfeki@sfu.ca
301325821

***Abstract*—Given a program, the automatic exploit generation (AEG) research challenge is to both automatically find bugs and generate working exploits [1]. The generated exploits clearly demonstrate that a bug is security-critical. Approaches for vulnerability detection and exploitation have historically involved manual effort through traditional penetration testing. The development of successful AEG techniques allows for automated vulnerability detection which can be utilized at a larger scale. This paper presents a prototype-level tool called Exploot. It detects a specific vulnerability in C binaries - the vulnerability to control-flow hijacking through a strcpy buffer overflow. A working exploit is then generated in the form of command-line input to the binary, which when fed to the executable, results in a control-flow hijack by launching the vim editor. The angr framework was utilized to perform static binary analysis in order to detect vulnerable strcpy method calls. We then use angr's symbolic execution capabilities to generate a working exploit.**

I. Motivation

Traditional methods of security testing, such as penetration testing and security audits, are time and resource-intensive. Although AEG does not promise to replace traditional security testing yet, it can be deployed in circumstances where traditional means are not possible due to lack of resources. AEG can also be used as a pre-preliminary to security audits by picking the lowest hanging fruit in terms of vulnerabilities that are relatively trivial to detect. Where Automated Exploit Generation shines is the detection specific types of vulnerabilities at a large scale. Moreover, security audits often rely on the existence of source code for the inspected software. If the source code is unavailable or inaccessible due to classified information, the alternative would be reverse engineering the binary code and analyzing the result. This alternative is often limited in its effectiveness, since important information is lost in the process of compiling source code into binary.

II. Existing Work

Blackbox Fuzzing [4] offers a promising path towards the automated detection of vulnerabilities such as buffer overflows as it requires no source code and can be fully automated. Vulnerability detection through mutation-based blackbox fuzzing typically works by having the fuzzing engine perform random mutations on well-formed program inputs until the input causes a crash when fed to the program. If the crash is a result of a buffer overflow, which is likely, then a buffer overflow vulnerability along with a concrete input that causes it is discovered.

III. Implementation

A. Prerequisites

The Exploot prototype is developed for the Linux Platform. So far, it only works on 32-bit executables and requires ASLR and Executable Space Protection to be switched off.

B. Tools

Exploot was developed using angr - a python framework for analyzing binaries [8]. The `strcpy_find` example from the angr-docs repo [8] repository was used as a starting point for the tool's development.

C. Vulnerability Detection

Exploot detects vulnerabilities in program according to the following definition of "vulnerable":

The program must satisfy two conditions.

1) The analyzed program binary must accept user input. The Exploot prototype is limited to generating exploits for programs that accept a single token of command-line user input
2) There exists a path in the program's control-flow graph (CFG) towards a strcpy method call such that the command-line input is written into the destination buffer parameter of the strcpy method.

Given an executable that accept a single token of command-line user input, Exploot first recovers the CFG of the program. It then performs a static analysis on the recovered CFG with the goal of finding the address of the strcpy function in memory. The found strcpy function address is then used by Exploot in a dynamic symbolic analysis (concolic analysis) to detect when execution hits a strcpy method call.

Once a strcpy method call is reached in the concolic analysis, the symbolic constraints at that point of the execution are then merged with another symbolic constraint that captures the buffer-overflow vulnerability of that specific call to strcpy. That other symbolic constraint is a simple requirement that that the strcpy source buffer parameter be equal to the commandline input. Note that both the source buffer parameter and commandline input are represented by symbolic values that must adhere to all symbolic constraints. By simply solving for the "satisfiability" of the augmented set of constraints, we can determine if our symbolic execution has reached a buffer overflow-vulnerable strcpy method call. More specifically, if the set of constraints are satisfiable, then there

exists a command-line input to the binary which will cause a buffer overflow.

D. Exploit Generation

If Exploot determines that a vulnerable strcpy method call has been reached, it can then start extracting the current stack frame's run-time information and use that to generate an exploit input string which will be fed to the executable. The specific run-time information that will be of use when generating an exploit are the address of the buffer susceptible to overflow, as well as the address of the return address, at which the attack aims to inject the address of injected shellcode. If a Canary Value protection scheme is used, the exploit string must take into account that the Canary Value should be changed on the stack after the buffer overflow. Therefore the contents of the stack between the buffer and the return address will be the third piece of run-time information to extract from the current stack frame. The algorithm used to generate the exploit constraint is referenced from [1]. The challenge in exploit generation lies in having the generated exploit string satisfy the following 3 conditions:

1) It must contain shellcode consisting entirely of alphanumeric characters. This is necessary in case the binary only accepts alphanumeric command-line input.
2) The execution of the vulnerable strcpy must overwrite the location of the return address on the stack with the address of the shellcode injected in condition (1).
3) To avoid overwriting a Canary Value, memory contents on the stack between the buffer and the return address should remain the same after the exploit string is written into the overflowed buffer.

```
// input : (bufaddr, &retaddr, m)
// output: exploit constraint
    for i = 1 to len(m) do
        exp_str[i] m[i] ; // restore Canary Value
    offset = &retaddr − bufaddr;
    jmp_target = buffadr + offset + 8 ;
    exp_str[offset] = jmp_target ; // eip hijack
    // exploit constraint to be appended to the SymEx state
    for i = 1 to len(shellcode) do
        exp str[offset + i] shellcode[i];
    return (Mem[buffaddr] == exp_str[1]) & ... &
    (Mem[buffaddr+len(m)] == exp_str[len(m)]) ;
```

Listing 1. Exploit Constraint Generation

The constraint generated by the function is Listing 1 is then appended to the set of constraints at the current point of execution - right before the vulnerable strcpy call instruction gets executed. Then, by using angr's built in SMT solver to solve for the value of the symbolic variable representing the command-line user input, the result is an exploit string which when fed to the executable will hijack the control flow. The example found in here illustrates the an exploit which spawns the vim text editor.

E. Guessing the Jump Address

Angr uses the unicorn CPU emulator framework [5], the addresses used in a non-emulated program execution are not guaranteed to be consistent with that of an emulated execution. The implementation of Exploot must therefore account for this discrepancy. The aim is that within the exploit string generated by Exploot is the address of the first shellcode instruction to which execution is meant to jump. This is the address that overwrites the original return address in the attack. However, in a non-emulated execution of the vulnerable program, the address of the shellcode injected through the attack is unknown due to the discrepancy mentioned earlier. When generating an exploit string, an educated guess must be made as to which address the shellcode starts at in memory. Exploot attempts to overcome this challenge through a generous use of nop slides and an approximate calculation of the actual shellcode address during non-emulated execution.

1) Nop slides: A nop slide is a technique commonly used to "slide" the CPU's instruction execution flow to a desired location [5]. If control flow lands anywhere on the slide, program execution will semantically remain the same as the control flow slides to the first non-nop instruction. This is akin to having the first instruction of a sequence of instruction in memory possess multiple addresses by which it can be referenced.

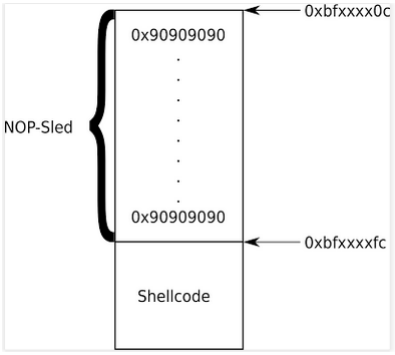

Fig. 1. The anatomy of a Nop Slide in an upwards growing stack

It is important to note that in the shellcode address approximation, an approximated address that is too low in the stack - lower than the actual shellcode address - is desired over one that is too high. This is due to the generous use of nop sleds. An address that is too low can be remedied by landing anywhere on the injected nop sled, given a nop sled that is large enough. Since our implementation can only inject the nop slides and the shellcode lower on the stack than the return address, having control flow jump to an address that is higher on the stack than the original return address cannot be saved by a nop sled, as the nop sled will always start after the return address in memory.

2) Address Calculation: The approximation is based on calculating the offset of the vulnerable stack frame - the frame containing the vulnerable strcpy method call - with respect to the main function stack frame in the emulated program execution. The address of the main function stack frame in a non-emulated run is then extracted and summed with the previously calculated offset, obtaining an approximation of where the vulnerable stack frame should lie within memory in a non-emulated execution of the program. This approximation method relies on the assumption that the distance between the vulnerable strcpy frame and the main function frame remain the same - or at least do not differ significantly - between the emulated and non-emulated execution of the program.

IV. Challenges

1) Stack Padding: Exploot was originally intended to work on both the osX platform as well as Linux. However, the subtle discrepancies in memory management between Mach-0 binaries and ELF binaries had made this challenging. The following 2 examples demonstrate an identical control flow-hijack on both platforms [9].

```
#include <stdlib.h>
#include <stdio.h>
void function() {
    char buffer1[5];
    int *ret;
     ret = buffer1 + 13;
     (*ret) += 7;
}
void main() {
  int x = 0;
  function();
  x = 1;
  printf("%d\n",x); \\ prints 0
}
```

Listing 2. Simple control-flow hijack on osX

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

void function() {
    char buffer[5];
    int *ret;
     ret = &ret + 3;
     (*ret) += 7;
}
void main() {
  int x = 0;
  function();
  x = 1;
  printf("%d\n",x); \\ prints 0
}
```

Listing 3. Simple control-flow hijack on Linux

2) Null bytes in the shellcode: In order for a buffer overflow attack to be successful, the shellcode must not contain any null bytes. The is because upon encountering a null byte in the source buffer of a strcpy method call, the null byte is regarded as the end of the buffer, which is not the intention of the attacker. It is possible to ensure the absence of null bytes in both the instruction bytes in the shellcode and the part of the shellcode corresponding to contents between the buffer and the return address. However, it becomes a challenge when the address of the shellcode on the stack contains a null byte, since the address of the shellcode is itself part of the shellcode.

Possible remedies to this include utilizing more advanced exploit techniques, such as heap spraying. Spraying the heap in this case would consist of having the shellcode lie in an address of our choice in the heap - overcoming the null byte constraint - and preceding it by a giant nop slide in order to increase the likelihood of the exploit working.

Another common workaround is a Return-to-libc attack. With the shellcode no longer requires injection into program memory. One could carefully craft a ROP gadget such that the first address jumped to does not contain any null bytes.

In the prototype of Exploot, it was found that 32 bit binaries have their stack start at addresses high enough not to contain any null bytes - at least in the most significant bytes of the address. Confining Exploot to work with 32-bit binaries was a quick and dirty workaround for the null byte challenge.

V. Evaluation

A significant amount of subtly was encountered when trying to have Exploot generate a working exploit. As a result, various corners have been cut to get a minimal implementation working.

A. Non-alphanumeric characters in the shellcode

Since the exploit string generated by Exploot is meant to be fed into the binary as a command line input, it is necessary that the values of the bytes correspond to values that have ASCII representations. The Exploot prototype does not address this issue yet, which prevents a rigorous evaluation of the tool.

B. Failure to work outside gdb

Due to the non determinism in the memory layout of binaries executed outside gdb, where the address of the stack cannot be clearly printed out which could help gain a clear picture of memory management, the generated exploit for evaluation purposes only works within gdb. The reason for non-deterministic behaviour of the memory layout outside gdb could be attributed to environment variables.

For the purposes of uncovering a vulnerability, having an exploit work in gdb seems to be an effective enough proof for the presence of the vulnerability, since all it would take to carry the exploit outside gdb is a better understanding of the memory layout in binary executions outside gdb.

VI. Future Research

Many of the discussed challenges are not addressed in the current implementation of Exploot. Addressing those issues would be a first step in future work on Exploot.The current implementation does not continue traversing the CFG if the constraint during exploit constraint generation cannot be satisfied. Moreover, the implementation should consider having the executable take in more than a single token of input, as most realistic command-line tools do.

References

[1] Avgerinos, Thanassis, et al. "Automatic exploit generation." Communications of the ACM 57.2 (2014): 74-84.

[2] Cha, Sang Kil, et al. "Unleashing mayhem on binary code." 2012 IEEE Symposium on Security and Privacy. IEEE, 2012.

[3] Wang, Yan, et al. "From proof-of-concept to exploitable." Cybersecurity 2.1 (2019): 12.

[4] Godefroid, Patrice, Michael Y. Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing." Queue 10.1 (2012): 20-27.

[5] Quynh, Nguyen Anh, and Dang Hoang Vu. "Unicorn-The ultimate CPU emulator." (2015).

[6] corelanc0d3r (December 31, 2011). "Exploit writing tutorial part 11 : Heap Spraying Demystified". Corelan Team. Archived from the original on 25 April 2015. Retrieved 15 January 2014.

[7] Kyle Ossinger (October 1, 2018). https://github.com/angr/angr-doc/blob/master/examples/strcpyfind/solve.py

[8] Shoshitaishvili ,et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis." IEEE Symposium on Security and Privacy (2016)

[9] Aleph, One. "Smashing the stack for fun and profit." http://www. shmoo. com/phrack/Phrack49/p49-14 (1996).