2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT) | 978-1-7281-8595-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCCNT51525.2021.9579959

AngErza: Automated Exploit Generation

Shruti Dixit[1], T K Geethna[1], Swaminathan Jayaraman[1], and Vipin Pavithran[2]

[1]Department Of Computer Science and Engineering
[2]Amrita Center for Cybersecurity Systems and Networks
Amrita Vishwa Vidyapeetham, Amritapuri, India
dixitshruti09@gmail.com, geethna.teekey@gmail.com, swaminathanj@am.amrita.edu, vipinp@am.amrita.edu

***Abstract*—Vulnerability detection and exploitation serves as a milestone for secure development and identifying major threats in software applications. Automated exploit generation helps in easier identification of bugs, the attack vectors and the various possibilities of generation of the exploit payload. Thus, we introduce AngErza which uses dynamic and symbolic execution to identify hot-spots in the code, formulate constraints and generate a payload based on those constraints. Our tool is entirely based on angr which is an open-sourced offensive binary analysis framework. The work around AngErza focuses on exploit and vulnerability detection in CTF-style C binaries compiled on 64-bit Intel architecture for the early-phase of this project.**

***Index Terms*—Automated Exploit Generation(AEG),buffer overflow, angr, Symbolic Execution**

I. Introduction

Software vulnerabilities arise due to many factors that includes design flaws, programming defects and configuration errors. If not detected and rectified, the software faces the risk of exploitation leading to potential security breaches which can prove to be costly. Consequently, identifying vulnerabilities in the code has become indispensable in secure software development. During this phase, major threats in software applications are detected by techniques such as code auditing, fuzzing applications, threat modelling and application penetration testing.

Vulnerabilities are of different types and detection techniques vary with each type. Detection involves analysis of code, either source or binary. For instance, a buffer overflow arises when the size of a buffer in memory is less than the size of the data is being stored in that buffer. This vulnerability can be exploited by an attacker overwriting memory address for example, the return address in a call stack to execute the malicious code of the attacker.

The process of detection is complex and many vulnerabilities go unnoticed during manual detection. Also, for crafting software vulnerability exploits one requires comprehensive knowledge of the underlying system like file format, processor architecture, mitigation enabled on the software, and working of the host operating system. Thus, source code analysis alone would be an incomplete approach for the same. The focus of our work is on dynamic binary analysis and exploit generation, wherein the raw binaries that compose a complete application are analyzed during run-time. This is especially helpful when there is no or limited access to source code.

AngErza is a solution for developers to validate their code and also to understand in what ways the program can be vulnerable. It generates automated exploit payloads for 64-bit ELFs x86 architecture with stack buffer overflow vulnerability. AngErza detects buffer overflow and format string vulnerabilities by doing dynamic analysis on the binary using r2pipe. It checks for constraints in the program and uses angr [9]–[11] to craft an exploit payload for attacking buffer overflow. For the initial stage AngErza focuses on CTF-style binaries. CTF (Capture the Flag) competitions aim to teach learn secure coding practices in a gamified manner. The vulnerable binaries used in such competitions are usually less obfuscated compared to real world applications. AngErza can be used by CTF players and developers to test the security of binaries and implement more checks by understanding the severity of each bug.

The main contribution of the paper is AngErza, a fully automated tool based on dynamic symbolic execution and analysis that analyzes the vulnerability of GNU compiled 64-bit binaries for the bugs of buffer overflow class. Through a variety of experiments, we show how the AngErza not only detects overflow vulnerability, but also determines the length and severity of each bug. The rest of the paper is structured as follows. Section II provides a brief summary of the related work. Section III discusses the core of our work on automated exploit generation for buffer overflows. Section IV talks presents the results of AngErza and analysis. Section V provides our conclusions and summarizes the directions for future work.

II. Related work

There has been considerable interest in program analysis, vulnerability detection, and automated exploit generation for secure software development. In this section we discuss some of the closely related work.

Program analysis involves checking the program for its correctness and to rule out any unintended bugs. The following are the two techniques for performing program analysis:

Static Analysis involves analysis at the source code level for any existing vulnerabilities without running the program. A data flow, control flow, and lexical analysis is done for the program to find any vulnerabilities. However, relying on static analysis alone maybe insufficient as this technique misses out finding vulnerabilities/bugs which could exist only during run-time execution.

Dynamic Analysis requires the program to be executed for vulnerability detection and testing. It involves debugging the code manually or using automated tools for checking memory corruption errors, code coverage, and other possible run-time errors. Dynamic analysis helps discovering vulnerabilities which maybe complex to detect during static analysis. But it might miss out on paths which may not be covered during run-time.

Dynamic analysis can be combined with visualization methods to provide structural and semantic summarization [12], [13] of program behavior. These can serve as effective aids in debugging design errors [14].

Vulnerability Detection in Binaries. The two most widely adopted techniques for vulnerable detection in binaries are fuzzing and symbolic execution.

Fuzzing [8] is an automated vulnerability detection technique in which random data is sent to the program so as to result the program in unexpected behaviour like getting exceptions, program crash or undefined output (especially memory leaks). This technique is widely used in software vulnerability detection as it is a simpler way of finding whether a bug exists in an application. However, sanitizers are used to assess the sensitivity of the inputs which lead to any undefined behaviour of the application. Also, a "dumb" fuzzer lacks generating input which helps covering all paths of a program.

Symbolic Execution [7] is a technique which helps determining an input that could lead to execution of a conditional branch or a specific path in the program. In the field of vulnerability detection this technique can be used to find an input which could satisfy constraints for triggering undefined behaviour in the application. However, this technique suffers from path explosion when finding all possible paths in a complex program.

Automated Exploit Generation. Over the last decade there have been considerable research work on automated exploit generation. As discussed earlier exploit generation requires understanding all the aspects of the applications and its environment in detail. The initial approach introduced in paper titled AEG: Automatic Exploit Generation [1] used was by using both source code and run-time binary information. However, this information may not be available for all vulnerability detection scenarios. Symbolic execution was used in the later approaches. The paper AEG for Buffer Overflow Vulnerabilities [2] presents a solution that relies on the binary code for the binary analysis and helps bypassing mitigations such as ASLR (address space layout randomization) and non-executable stack. The tool relies on angr for symbolic execution but suffers from path explosion in complex programs. A hybrid approach however solves this and helps in easier bug detection and exploration. Mayhem [5] which uses a novel hybrid-symbolic execution technique and covers two classes of bugs: buffer overflows and format strings. PolyAEG [3] generates automatically generates multiple exploits for a vulnerable program and thus is more resilient. However, it does not bypass ASLR and suffers performance issues due to reliance on symbolic execution. Overall, AEG has been hot-topic for research continuously and has been widely been used in software testing and validation.

Considering the above approaches we have applied symbolic execution in AngErza for solving constraints for generation of exploit payload. It does so by using angr which is a binary analysis framework that uses dynamic symbolic execution for analyzing binaries and has been at the forefront of automated analysis. We dynamically analyze the binary using r2pipe for detecting buffer overflow and format string. In the coming section we have discussed our approach in detail.

III. AngErza

This section would talk about the working of AngErza in detail. AngErza is a light-weight approach for code authors for testing their programs. If any vulnerability is found in the program the developer can analyze the generated exploit and introduce checks/patches in the program to harden the code against any exploits.

The main driver script for AngErza is written in python 3.6. As discussed earlier AngErza relies on binary analysis framework in python called angr for all its functionalities. angr helps performing dynamic symbolic execution and static analysis on the binary. The functions of AngErza is divided into 2 sub-modules: finding buffer overflow and length, and exploit generation. If buffer overflow bug is not detected in the binary then AngErza checks if there is a format string bug. We will be taking a simple buffer overflow scenario and walking step-by-step through the above modules of the tool. The below code, bug.c, is compiled without any stack protectors.

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

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

  char buffer[16];
  read(0,buffer,80);
  return 0;
}
```

A. Terms and definitions

Definition 1. Format string vulnerability : a program vulnerability when the user's input is treated as format string or in other words, when the format specifier and the parameters to be printed do not match. An attacker can do memory reads and writes by controlling this vulnerability.

Definition 2. Buffer overflow : when the data sent to a program is stored in a buffer whose size is less than then input size.

Definition 3. r2pipe : an API that allows users with methods that can be used to send and run r2 (radare2) commands on a binary. Radare2 is a framework which is used for disassembling, patching, and debugging for reverse engineering binaries.

Definition 4. Exploit: an exploit is a well crafted string or code which attacks the vulnerabilities in a program to cause unintended behaviour in the program.

B. Determining buffer overflow

Using angr first all the functions which are called in the binary are listed out. In the above list of functions AngErza checks for the stdin functions. In our case the stdin function used is *read*. Then it does a dynamic analysis using r2pipe APIs where the arguments of stdin functions are checked. This is done by getting the register values using r2pipe which store the arguments that are passed to the function. Buffer size is calculated as the difference of the stack address and the base pointer of stack (RBP). If the buffer size is less than the size of input then the constraint for overflow is satisfied. The difference of the size of input and buffer size is evaluated as the overflow size. Here, the size of the buffer is 16 whereas the input size is 80. Thus, the buffer overflow bug exists here and the size of which is 64.

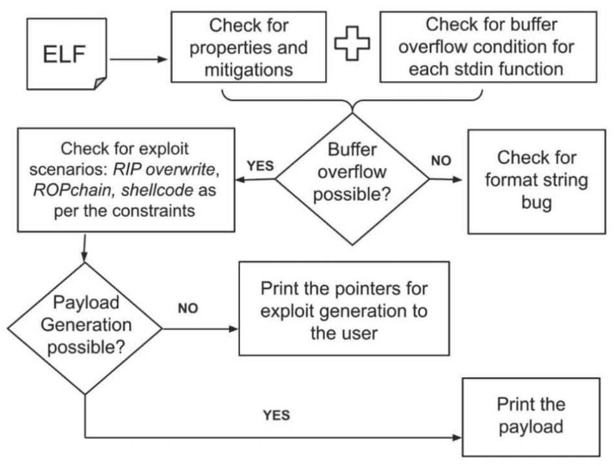
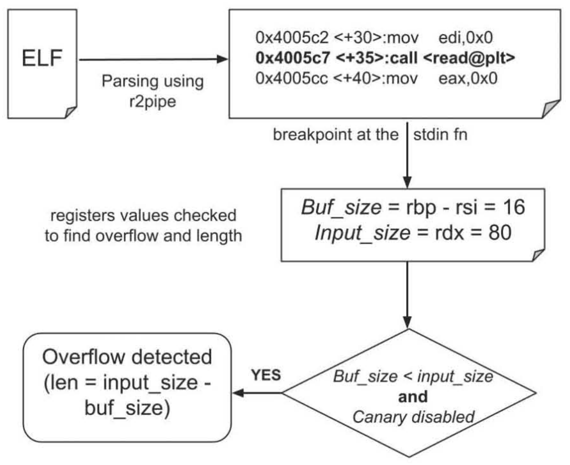

ELF
Check for properties and mitigations
Check for buffer overflow condition for each stdin function
Check for exploit scenarios: RIP overwrite, ROPchain, shellcode as per the constraints
YES
Buffer overflow possible?
NO
Check for format string bug
Payload Generation possible?
NO
Print the pointers for exploit generation to the user
YES
Print the payload

Fig. 1. overall working


ELF
Parsing using r2pipe
breakpoint at the stdin fn
registers values checked to find overflow and length
Buf_size = rbp - rsi = 16
Input_size = rdx = 80
Buf_size < input_size and Canary disabled
YES
Overflow detected (len = input_size - buf_size)

```
0x4005c2 <+30>:mov    edi,0x0
0x4005c7 <+35>:call <read@plt>
0x4005cc <+40>:mov    eax,0x0
```

Fig. 2. Detecting buffer overflow in bug.c

C. Find exploit payload

Next, we check if canary is enabled in the binary by getting the properties of the binary by using pyelftools. If canary is disabled in the binary then we can do attack which hijacks the return pointer. We check for the next mitigation which is PIE (position independent executable), by enabling this address space of the sections in the program are randomized each time. If this is enabled then we may only be able to do shellcode injection attack. For this, we must check if stack is non-executable (NX) or not. If NX mitigation is disabled then we can do the shellcode injection attack.

Considering the above constraints we check for the possibility of payload generation in any of the scenarios:

ROP chain generation : AngErza try to generate a ROP chain for doing an execve syscall. It achieves this by using angrop which is an angr based ROP chain generation tool. If the necessary gadgets are found in the binary for doing the execve syscall the ROP chain is generated else the possibility for the next attack is checked.

RIP overwrite : AngErza checks for the possible scenario of overwriting the return address of the current function, that is stored on stack, directly with a function such as system provided it is called in the binary. We achieve this by giving constraints to angr to generate an input such that the return address is overwritten.

Shellcode Injection : If NX is disabled, AngErza tries to find an appropriate shellcode matching satisfying the constraint of size as per the buffer size calculated.

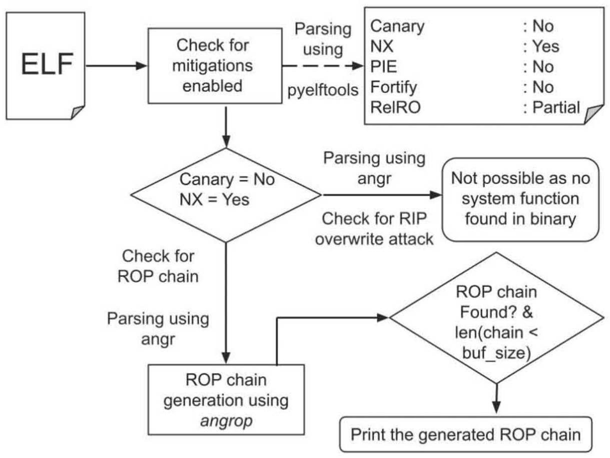

ELF
Check for mitigations enabled
Parsing using
pyelftools
Canary : No
NX : Yes
PIE : No
Fortify : No
RelRO : Partial
Canary = No
NX = Yes
Parsing using angr
Check for RIP overwrite attack
Not possible as no system function found in binary
Check for ROP chain
Parsing using angr
ROP chain generation using angrop
ROP chain Found? & len(chain < buf_size)
Print the generated ROP chain

Fig. 3. Exploit Generation of bug.c

If any of the attack payload is found by AngErza, it is printed out to the analyst which can be verified by testing it on the binary. For our example, bug.c AngErza was able to generate a ROP chain for execve syscall which we later verified to spawn a shell.

D. Finding format string bug

If the buffer overflow bug is not detected in the binary then AngErza checks for a format string bug but we do not generate an exploit for it.

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

int main(int argc, char const *argv[])
{
  char a[10];
  scanf(”%10s”,a);
  printf(a);
  return 0;
}
```

In the above example, the format string vulnerability exist in *printf(a)* as the user input is not formatted when printing. Similar to the buffer overflow detector, we use r2pipe API to set a breakpoint at the printf function. We check the value of the registers and see whether the first argument is a readable writable address on the stack. If yes, then we print ”format string bug is detected” else ”format string bug is not detected” is printed to the user.

IV. Result and Analysis

This section details about the evaluation of AngErza. We tested AngErza with C binaries compiled on x86 64-bit architecture. We developed AngErza to support bug detection, constraint generation and exploit generation. We use angr tool to do symbolic execution on the given binary and to generate constraints. We generate file and structure related constraints by extracting debug and compilation details. We produce python based payloads using angrop (rop-gadget finder) and hardcoded shell-codes. These payloads can be used in the exploit script to spawn a shell in the given system. We evaluated AngErza on a Linux machine with Intel Core CPU with 1TB hard disk and 4 GB RAM.

The tables below summarizes the results of the test binaries which have been used to validate AngErza. The various constraints, bug and attack are mentioned against each binary. The results are denoted by TP - True Positive, FP - False Positive, TN - True Negative, FN - False Negative.

Vulnerability Detection and Constraint Generation. We use a test case to illustrate the process of exploit generation for the given vulnerable binaries. We are presenting it using three different case scenarios from the table above. The binary demo-win has a user defined function call to system, but the control flow doesn’t allow the binary to execute it. AngErza gets the file properties and protections, in this case NX bit is enabled and the tool successfully detects the overflow length. According to the overflow length and the protections, the tool tries to model a payload to spawn a shell. Here, it generates a return-to-system attack payload. For the second case of demo, the file protections forces the tool to model an exploit which involves making a call to execve() by setting the arguments and returning to a syscall gadget. The ROP-gadgets are generated with the help of angrop

TABLE I
BUFFER OVERFLOW DETECTED

Binary	Constraints	Bug	Attack	Result
demo win	system-function	gets	ROP chain	TP
demo	NX-enabled	gets	syscall ROP	TP
vuln	NX-disabled	read	return to shellcode	TP
lottery	statically linked	fgets	syscall ROP	TP
shellcode-golf	NX-enabled	read	return to shellcode	TP
Double Trouble	Fortify disabled, system-function	read, printf	format string, ROP chain	TN

and a chain is generated accordingly which will spawn a shell. In the shell binary, NX protection bit is disabled which makes the tool take another path which involves in generating a payload which will result in a shellcode. AngErza calculates the overflow and buffer length and prints out the appropriate shellcode, which can be used in the exploit script to facilitate the return-to-shellcode attack. The binary lottery was a statically linked 64 bit executable with an fgets function taking input. The overflow was detected by AngErza but it was not able to generate a complete exploit because of the size of the binary. Double Trouble binary had 2 bugs - buffer overflow and also format string vulnerability. AngErza detected detected the overflow and generated a ROP chain exploit for the overflow bug. But it was not able to detect a format string as it already generated an exploit for overflow vulnerability. All the other binaries used for testing AngErza on the basis of overflow detection are different variants of the protections and binary structure.

TABLE II
BUFFER OVERFLOW NOT DETECTED

Binary	Constraints	Bug	Attack	Result
format-string	Fortify-disabled	printf	format string attack	TP
no-bug	All protections enabled	NIL	no vulnerabilities	FN
controller	system function	fgets	integer overflow	TN
returning	fortify disabled	snprintf	format string	TP

In the format-string binary, a buffer overflow was not detected and hence there was no possibility of an injection attack. The binary passed checks for a possible format string attack and printed out the result. The binary no-bug has no vulnerable functions/processes. Hence it was not able to detect an overflow or a format string attack. AngErza displays that no vulnerabity was discovered. The controller binary had a hidden integer overflow bug which unwraps during runtime. AngErza was not able to detect this possibility as the maximum size does not lead to an overflow but the possibility of a negative integer was overlooked. In the case of returning, AngErza was able to detect the format string vulnerability in the snprintf function which can lead to memory leak and memory overwrite.

V. CONCLUSION

In this paper, we presented AngErza, a fully automated tool based on dynamic symbolic execution and analysis, which helps to analyze the vulnerability and exploitability of GNU compiled 64-bit binaries in the case of buffer overflow class of bugs. The tool was able to detect overflow vulnerability and the length of each overflow and its severity. AngErza was also able to detect format string vulnerability in buggy binaries. Using AngErza, we were able to evaluate the security of the binary and facilitate adding more checks. We have presented an evaluation of our tool and thus we are publishing it as a CTF-helper tool.

As AngErza relies on angr, which uses symbolic execution, it suffers from path explosion when working with obfuscated and large binaries. AngErza cannot bypass all binary mitigations and OS defenses (like ASLR). Future work for AngErza includes extending it to be resilient against these defenses. And also, enhance it to detect and generate exploits for other classes of bugs in both stack and heap. AngErza is limited to generating 3 types of attacks for exploiting overflow as explained in this paper. Thus. it can be improved to include other attacks such as return-to-libc and stack pivoting.

REFERENCES

[1] Avgerinos, Thanassis and Cha, Sang and Hao, Brent and Brumley, David. (2011). AEG: Automatic Exploit Generation.. Communications of the ACM. 57. 10.1145/2560217.2560219.

[2] L. Xu, W. Jia, W. Dong and Y. Li, "Automatic Exploit Generation for Buffer Overflow Vulnerabilities," 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Lisbon, 2018, pp. 463-468, doi: 10.1109/QRS-C.2018.00085.

[3] Wang M., Su P., Li Q., Ying L., Yang Y., Feng D. (2013) Automatic Polymorphic Exploit Generation for Software Vulnerabilities. In: Zia T., Zomaya A., Varadharajan V., Mao M. (eds) Security and Privacy in Communication Networks. SecureComm 2013. Lecture Notes of the Institute for Computer Sciences,

Social Informatics and Telecommunications Engineering, vol 127. Springer, Cham. https://doi.org/10.1007/978-3-319-04283-1_14

[4] Gadient, Austin and Ortiz, Baltazar and Barrato, Ricardo and Davis, Eli and Perkins, Jeff and Rinard, Martin. (2019). Automatic Exploitation of Fully Randomized Executables.

[5] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleash-ingMayhem on binary code,"Proceedings - IEEE Symposium on Securityand Privacy, pp. 380–394, 2012.

[6] Brooks, T. (2017). Survey of Automated Vulnerability Detection and Exploit Generation Techniques in Cyber Reasoning Systems. ArXiv, abs/1702.06162.

[7] James C. King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385–394. DOI:https://doi.org/10.1145/360248.360252

[8] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. Commun. ACM 33, 12 (Dec. 1990), 32–44.

[9] Y. Shoshitaishvili et al., "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," 2016 IEEE Symposium on Security and Privacy (SP), 2016, pp. 138-157, doi: 10.1109/SP.2016.17.

[10] Shoshitaishvili, Yan and Wang, Ruoyu and Hauser, Christophe and Kruegel, Christopher and Vigna, Giovanni, Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware,NDSS,2015

[11] Stephens, Nick and Grosen, John and Salls, Christopher and Dutcher, Audrey and Wang, Ruoyu and Corbetta, Jacopo and Shoshitaishvili, Yan and Kruegel, Christopher and Vigna, Giovanni, Driller: Augmenting Fuzzing Through Selective Symbolic Execution, NDSS, 2016.

[12] S. Jayaraman, B. Jayaraman and D. Lessa, "Compact visualization of Java program execution," Software: Practice & Experience, John Wiley & Sons Inc., vol. 47, pp. 163-191, 2017.

[13] A. A. Aziz, M. Unny, S. Niranjana, M. Sanjana and J. Swaminathan, "Decoding Parallel Program Execution by using Java Interactive Visualization Environment (JIVE): Behavioral and Performance Analysis," in proceedings of 3rd International Conference on Computing Methodologies and Communication (ICCMC), pp. 792-797, 2019.

[14] K. P., J, Jayaraman, S, Jayaraman, B, M, S., "Finite-state model extraction and visualization from Java program execution," Softw Pract Exper. 51: 409– 437, 2021.

[15] V. S. Rao, T. Gupta, S. Vasan and L. R. Deepthi, "PHPIL: Fuzzing the PHP Interpreter with Custom Bytecode," 2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT), 2020, pp. 1-5, doi: 10.1109/ICCCNT49239.2020.9225578.

[16] author Seshagiri, Prabhu and Vazhayil, Anu and Sriram, Padmamala, AMA: Static Code Analysis of Web Page for the Detection of Malicious Scripts,Procedia Computer Science,vol. 93,768-773,doi: 10.1016/j.procs.2016.07.291

[17] Prakash, R. and Amritha, P. and Sethumadhavan, M.,Opaque Predicate Detection by Static Analysis of Binary Executables,pages. 250-258, isbn = 978-981-10-6897-3,doi= 10.1007/978-981-10-6898-0_21