Analysis of Automated Exploit Generation programs

Laasya Raagyni Kothapalli CS20B043

Mentor: Prof. Chester Reberio

Abstract

In recent times, the increasing reliance on technology and the complexity of software applications pose serious security threats. While progress has been made in security and binary analysis, limitations persist due to reliance on human expertise. This highlights the need for automating binary analysis and vulnerability detection. Several tools automatically identify vulnerabilities and generate exploit strings for binaries. However, many current Automated Exploit Generation (AEG) solutions overlook key security mitigation like ASLR, NX, or PIE implemented by modern operating systems to protect the binaries against buffer overflow, ret to libc attacks etc. We explore methods and the challenges for constructing an automated exploit generation tool. We initially explore different methods of binary analysis and vulnerability detection. We also briefly mention various AEG tools developed along with their limitations. The discussion also addresses challenges posed by modern security measures and explores advancements in automated vulnerability detection and exploit generation by particularly analysing the current best performing AEG programs.

1 Introduction

There are certain challenges involved in building an automated exploit generation tool and the task of building an AEG tool is divided into these components:

1) Vulnerability detecting techniques

2) Techniques for bypassing vulnerability mitigation and generating exploits

3) Building an automation tool

Initially, our focus revolves around various methods for vulnerability detection and binary analysis, as a considerable amount of research and tools were initially developed in this domain. In the early stages of exploring Automated Exploit Generation (AEG), significant attention was directed towards the practical need for AEG - detecting vulnerabilities in binaries to enhance their security and subsequently evaluating the potential of these vulnerabilities by generating exploits.

However, these methods often overlook the security mitigations implemented in modern systems. Recent tools have adopted a specific approach, emphasizing the detection and exploitation of vulnerabilities while considering the implemented security features. While these tools excel in detecting vulnerabilities in simple binaries, such as those used in Capture The Flag (CTF) challenges where leaks are intentionally provided in the standard output, they face challenges in addressing a broader range of applications involving complex and multi-threaded processes. Subsequent sections provide a brief overview of some implemented tools, with a detailed exploration of Zeratool - a current open-source AEG tool that effectively bypasses security mitigations to exploit binaries.

2 Binary analysis and Vulnerability detection

Despite ongoing efforts, software defects have always existed. Ensuring the safety, security, and reliability of software is a primary focus for many security researchers, leading to a significant emphasis on binary analysis. This is necessary because source code analysis alone is not comprehensive; combining it with binary analysis is essential. Moreover, in numerous cases, only binaries are accessible, particularly when examining embedded firmware, custom operating systems, and malware.

Challenges in binary analysis: The abstractions provided by the programming languages like the data types and data structures make it easier to understand the paths of execution driven by the data and the inputs. However, the missing of these abstractions make binary analysis challenging.

However, there exist distinct advantages to conducting binary analysis, including: 1) The inclusion of platform-specific details exclusively accessible during execution time. 2) Incorporation of critical information such as memory layout, register usage, and execution order, which is important in identifying various common vulnerabilities, such as memory corruption and buffer overflows.

2.1 Types of Vulnerability present:

Most commonly found vulnerabilities are buffer overflows, format string attacks and memory corruption vulnerabilities. These pose a risk of compromising application safety, granting attackers access to sensitive data, or seizing control of an application's flow to execute programs at their discretion. Consequently, many automated vulnerability tools prioritize the detection of these specific vulnerabilities.
Buffer Overflows: A buffer overflow vulnerability occurs when a program writes more data to a block of memory, or buffer, than it was allocated to hold. This overflow can lead to unpredictable behavior, crashes, or, in more severe cases, exploitation by attackers. By carefully crafting input to exceed buffer limits, attackers may overwrite adjacent memory and potentially execute malicious code, compromising the security and integrity of the affected software.
General Memory corruption: Memory corruption occurs when data in a previously allocated location is altered. Buffer overflow is one example for this. Other types of vulnerabilities that lead to memory corruption array index out of bounds, attempting to use a pointer that has been freed already or using an address before memory is allocated[5]
Format String Vulnerability: A format string vulnerability arises when user input is incorporated as the format argument in functions such as printf, scanf, or their counterparts.
The format argument encompasses diverse specifiers, providing an avenue for potential data leakage if an attacker gains control over the format argument passed to printf. Given that printf and similar functions are variadic, they extract data from the stack based on the specified format.
For instance, consider the format argument "$\%x.\%x.\%x.\%x$." This could compel printf to extract and print four stack values in hexadecimal, potentially disclosing sensitive information. Although these vulnerabilities can be potent, their occurrence has become increasingly rare in contemporary programming due to modern compilers issuing warnings when printf is called with a non-constant string[11].

2.2 Detecting vulnerabilities using Binary analysis:

1. **Static binary analysis:**
   This involves analyzing a binary without its actual execution, typically by loading and processing the binary to be analysed. The binary is parsed, and its assembly instructions are translated into an intermediate language. Subsequently, a Control Flow Graph (CFG) is generated. Control Flow Graphs illustrate the possible paths a program can take, where nodes signify basic blocks of instructions and edges denote points of control flow changes. CFGs are important in vulnerability detection, offering a systematic means to explore all potential paths within an application.

Limitations: This technique is constrained by performance and scalability costs. Challenges arise, particularly with indirect jump statements, as they are intricate to resolve. Indirect jumps transfer control to a target with a value arbitrarily calculated or dependent on input or the context of the application. To address these challenges, static binary analysis tools resort to approximations about the control flow. However, this approach introduces the risk of false positives or may overlook vulnerabilities due to incomplete control flow graphs.

2. **Dynamic binary analysis:**
This techniques examines the program behaviours while it is running in a given environment. The common dynamic analysis techniques are:

1) Fuzzing: This techniques is used to detect the vulnerabilities by attempting to crash the program over a large range of inputs. It is an example of concrete execution where it tries to find inputs that lets the binary perform unsafe operations.

Limitations: Since they generate a large range of inputs randomly they tend to fail to successfully pass through input processing code. Fuzzing cannot work on specific inputs (limited set of valid values that the binary considers) but it performs well on general input (wide range of valid values). Another limitation of fuzzing is that it cannot pinpoint the source of vulnerability in the binary which makes it difficult to exploit the vulnerability detected.

Numerous advanced fuzzing tools have been developed to overcome these limitations. Some of these are: LibFuzzer[13] and Markov Fuzz[14] specialize in coverage-guided fuzzing. On the other hand, Steelix[15] utilizes static analysis techniques to direct the fuzzing process. T-Fuzz[16], in contrast, employs dynamic symbolic execution to enhance its fuzzing capabilities.

AFLFast[14] enhances path coverage through optimized mutation strategies derived from AFL. In contrast, AFLGo[18] and Hawkeye[19] employ target-oriented fuzzing to concentrate on specific program areas. For a more comprehensive program analysis and improved defect detection, tools like Driller[6] and HFL[20] integrate symbolic execution with fuzzing.

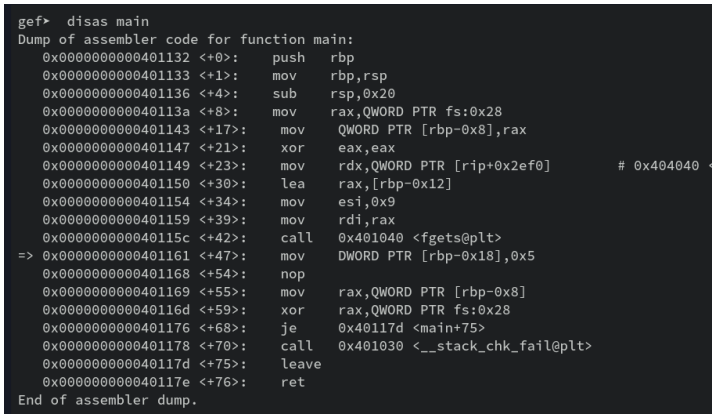
2) Symbolic execution: This techniques involves the usage of symbolic values as input instead of specific values like in general execution. During the execution both the memory and registers are tracked and are modeled symbolically. These symbolic values are used to generate path constraints which are logical formulas that represent program state and transformations between program state. The formulas are fed to a constraint solver or a satisfiability modulo theory solver (SMT). This solver helps in deriving inputs that drive the exploration of paths in the application. Mayhem[12] and S2E[17] are the first to apply these techniques to binary code. Dynamic symbolic execution is so powerful because it can trigger specific application states using learned path constraints, making it an ideal technique for discovering vulnerabilities in binary code.
Limitations: Dynamic symbolic execution has a problem called "path explosion," where new paths keep multiplying at each decision point. This makes it computationally expensive and limits how well it can scale for analyzing systems.

A modern solution to this problem is to mix both concrete (real) and symbolic (abstract) execution, a method known as concolic execution. Another approach combines dynamic symbolic execution with fuzzing to create a "guided" fuzzer, like Driller[6].This strategy employs dynamic symbolic execution to steer path exploration, tasking it with refining input and seamlessly integrating it back into the fuzzer. The goal is to minimize reliance on resource-intensive operations, such as dynamic symbolic execution, and favor more cost-effective techniques like fuzzing for improved code coverage in vulnerability assessments.

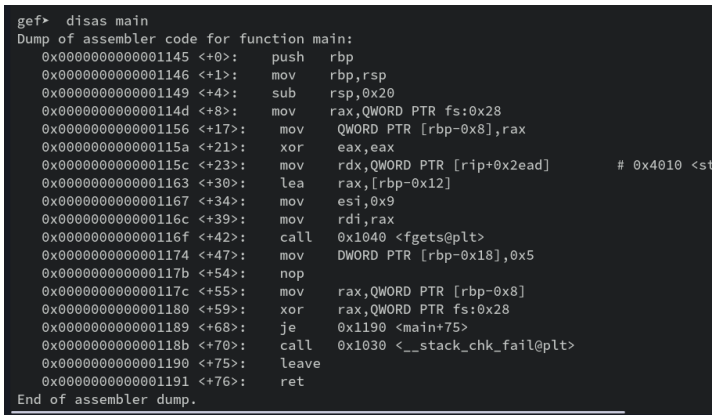
3 ASLR and Position independent executable (PIE)

ASLR, or Address Space Layout Randomization, is a mitigation technique designed to enhance the security of systems by making it more challenging for attackers to execute arbitrary code in the event of a memory corruption vulnerability, such as a buffer overflow. The core principle of ASLR involves introducing randomness into the memory layout of an executable, including the base address, heap, stack, and loaded libraries.Consequently, even if an attacker manages to control the flow of execution, like manipulating the instruction pointer, the location of the arbitrary code becomes unpredictable.

For effective ASLR implementation, support is required both in the operating system, primarily in the loader of executables, and in the executable itself. The executable should be compiled as Position Independent Code (PIC) and have support for relocations when absolute addressing is employed.
When binaries are compiled without Position Independent Executable (PIE) enabled, each instruction's address is absolute. Conversely, when binaries are compiled with PIE enabled, addresses are represented as offsets. This is achieved through the use of the Procedure Linkage Table (PLT) and the Global Offset Table (GOT).



(a)



(b)

Figure 1: Assembly code of main function a) without PIE b) with PIE enabled

4 Brief introduction of already developed tools

Heelan et al.[23] employ binary instrumentation for taint propagation and run-time information gathering. Exploits are generated by checking whether the EIP register is affected by the taint. AEG [3] pre-processes the source code to produce byte-code, leveraging conditional symbolic execution to identify vulnerable functions, overflow-covered objects, and bug-triggering paths. Simultaneously, dynamic binary analysis extracts run-time information, culminating in exploit generation. This approach is extended to Mayhem[12] for supporting binaries. CRAX[24], beginning at the crash point, symbolically executes the program to pinpoint exploitable states and generate corresponding exploits. Padaryan et al.[25] address ASLR, and Xu et al.[26] further extend the method to tackle NX. However, this solution, while considering NX, falls short when ASLR and NX are both active, relying on the program containing the "jmp esp" instruction to complete the exploit. Under the concurrent activation of ASLR and NX, Zeratool[1] successfully manipulates program control flow to the win function and executes ROP attacks. But it does not account for programs protected by PIE and relies on the presence of standard output functions (puts, printf, etc.) when using ROP attacks.

5 Implementation of the current performing tools

Most of the current developed automated exploit programs use certain symbolic execution engines like angr and binary analysis tools like radare2. **Vulnerability detection:** Like discussed before, symbolic executors assign symbolic values take symbolic values as the input. Even the memory and the registers are tracked symbolically. Hence for finding a vulnerability we try to get the path in which at one point the pc or the instruction pointer is symbolic. This implies that the instruction pointer is dependent on the input we give. If we want pc to be a particular value, we set the pc to be that value use a constraint solver for that path and we finally get the value of the input. This lets us target vulnerabilities like buffer overflow and format string vulnerabilities. **Exploit Generation:** Initially all binaries check if the win function is present (that contains the system function to open the shell or a leak to the flag). Using this address of the win function, they use the buffer overflow to transfer the control flow to this function. If the stack is executable then the programs writes the addresses of the system function and the arguments on the stack itself. The only difficulty here is to determine the address of the library functions like $system("/bin/sh")$ or $execve("/bin/sh")$.

5.1 Zeratool

Let's explore how Zeratool overcomes the challenges posed by ASLR to generate exploits effectively.

Zeratool stands out as an advanced open-source tool for automated exploit generation. Comprising 17 Python files, each serving a specific purpose, the tool lacks good documentation to understand the structure. To understand its intricacies, we delve into the program's detailed structure by examining the Python source code.

Zeratool excels in producing exploits for binaries compiled with ASLR, where the stack, heap, and library base addresses are randomized. Additionally, it proves effective in scenarios where binaries are executed remotely.
Inputs to the program: Binary, Libc file, URL and port for remote execution. Additionally we can provide inputs to only check overflow leak or format string leak.

5.1.1 Vulnerability detection:

Firstly, the program attempts to identify if a vulnerability is present by checking for the state in which the program counter (pc) is symbolic.

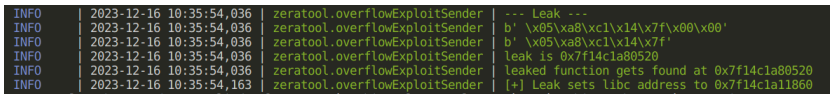
Subsequently, the program solves the constraints using a constraint solver for a given value of pc and checks the type of input obtained. For instance, if pc is assigned 'CCCCCCCC' and the input also ultimately contains 'CCC...' characters, then it indicates a buffer overflow. Similarly, if the input contains an element in a format string like $\%llx$, it signals a format string vulnerability.

(a)

Figure 2: Vulnerable path found using overflow vulnerability

A) If overflow vulnerability is present

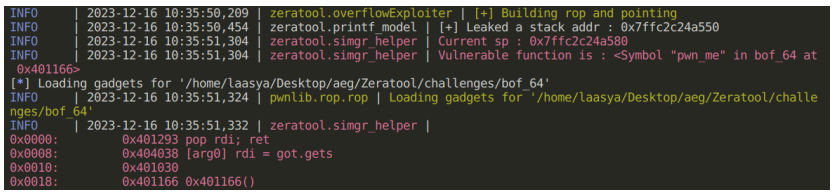
1. Identifies if the win function or a direct flag as a string is present:
   - This is identified by initially getting a list of all function details in JSON format using binary analysis with the tool radare2.
   - Checks for the list of all the functions that use system function (system@plt). This function will be present if this function is called somewhere in the binary. Hence the function that calls the system function would be our win function.
   - It analyses all the strings present in the binary which contain flag or pass.
   - Finally it gets all the functions that reference these strings.
   - All of these functions constitute our win function.
2. If the binary is has executable stack
   - If the binary has an executable stack it directly writes the instructions that execute the $system("/bin/sh")$ or $execve("/bin/sh")$ onto the stack and replaces the pc to this location where the instructions are present
3. If NX and ASLR are enabled:
   This means the stack is not executable and the addresses of stack, heap and the libraries are randomized.
   Hence we try to return to libc for opening up a shell. We initially try to find the base address of libc by leaking one of the functions in libc, we do this by:
   - Initially we load the libc library at 0x500000 location and then try to find a leak.
   - We try to get a leak through puts or printf functions, where the argument is the address of a got entry of a function in libc that has already been resolved. For example: If the function *gets* is already resolved, we try to print the got entry of gets by doing puts(got entry of gets). Here we build an rop chain using the address of puts@plt (say x) which is our return function, the argument to this is got of gets (say y) which is our leak function got address. The rop chain will be ⟨x (8 random bytes) y⟩.



(a)

Figure 3: Vulnerable path found using overflow vulnerability

- This way we leak the address of gets, we also need to make sure that we return to the vulnerable function finally since we need to get back to our system function from the leak. Thus the rop chain becomes ⟨x (address of vulnerable function) y⟩.
- We also need to find the gadget pop rdi; ret. If the binary is a 64 bit binary, the first argument it takes for a function is from rdi register. Thus pop rdi would put the value of got entry of the leak function in rdi register.



(a)

Figure 4: Vulnerable path found using overflow vulnerability

- In the previous step we get the vulnerable function address from a state where the pc was observed to be unconstrained. This is obtained by getting all the symbol (function name) addresses that are reachable from main (using the state) to achieve that pc unconstrained state. The vulnerable function address is the address of the symbol whose address is the highest addresses that is before the last basic block.
- We send the same string by connecting remotely to the binary and then based on the leak, we calculate the base address of libc from the offset.
- We now load our libc at that base address obtained in the previous step and we now build the rop chain for executing the system function for opening up a shell (we know the base address of libc)
- We send the same string to the remote library and thus this opens up a shell

b) If format string vulnerability is present

1. Using format string vulnerability we try leaking the addresses on the stack or form the got (.data field mostly). We do this by sending an exploit string of type $\%llx\%llx...\%llx$ and later replacing it by $\%llx\%(i+1)\$llx...\%(i+n-1)\$llx$ where n is the highest length of the symbolic part. With this we get various types of memory leaks and flags.
2. Using format string vulnerabilities we can overwrite locations we can access to any addresses, for example if a symbol A(say 0x7fff3456) is leaked then we can send $\backslash x7f\backslash xff\backslash x34\backslash x56\%(y)lx\%(i+2)\$n$ to overwrite the address pointed by 0x7fff3456 to 2+y.
3. Now with this we can overwrite any got entry with any address once the address of the got entry becomes known.

5.1.2 Limitations:

The tool has some existing bugs that need addressing to enhance compatibility with specific binaries. It currently struggles with Position-Independent Executable (PIE) enabled code, even in cases where a leak is directly present in the standard output of the binary. Despite these limitations, Zeratool has robust functionality, demonstrating effectiveness across a broad range of scenarios and it is one of the few tools that make use of format string vulnerabilities efficiently.

5.2 Faeg and Bof_aeg:

These recently developed tools share similar functionalities, employing comparable methods. They effectively operate even when both PIE and ASLR are enabled. While Faeg lacks publicly available source code, Bof_aeg provides its source code.

In most cases, Faeg outperforms Zeratool. But Faeg lacks the implementation to work for vulnerabilities other than buffer overflow.

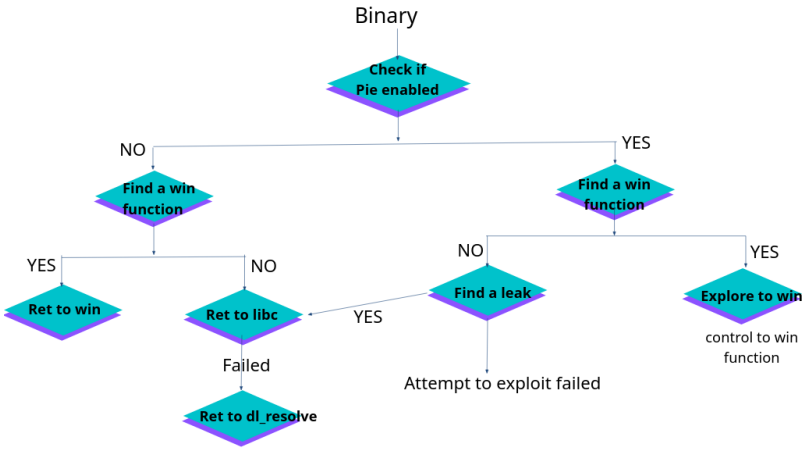
Binary name	Method branch	Security mitigation			Zeratool	FAEG
		ASLR	NX	PIE		
SWPUCTF2021_pwn	*Ret to shell*	Y	Y	N	**5.59s**	**1.08s**
csictf2020_int	*Ret to shell*	Y	Y	N	✗	**2.67s**
HNCTF2022_overflow	*Ret to shell*	Y	Y	N	✗	**3.42s**
TourCTF_pwn	*Ret to shell*	Y	Y	Y	✗	**8.72s**
NewStarCTF2022_gift	*Ret to libc*	Y	Y	N	✗	**1.34s**
sharkyctf2020_away	*Ret to libc*	Y	Y	Y	✗	**2.23s**
NewStarCTF2022_pwn	*Ret to libc*	Y	Y	N	**7.81s**	**1.51s**
SWPUCTF2021_white	*Ret to libc*	Y	Y	N	✗	**3.73s**
pbCTF_pwn	*Ret to libc*	Y	Y	N	**21.90s**	**3.97s**
KniCTF_cat	*Ret to csu*	Y	Y	N	✗	**5.13s**
gyctf2020_borrowstack	*Ret to stack pivot*	Y	Y	N	✗	**1.99s**
Metctf_duck	*Ret to dlresolve*	Y	Y	N	✗	**1.39s**
nahamCTF2021_smol	*Ret to dlresolve*	Y	Y	N	✗	**2.15s**
TFCCTF_resolve	*Ret to dlresolve*	Y	Y	N	✗	**1.51s**
HNCTF2022_pivot	***FAIL***	**Y**	**Y**	**N**	✗	✗

(a)

Figure 5: Performance of Zeratool over Faeg obtained from their research paper[3]

Faeg works in three steps:
1) Preprocessing stage - Preprocessing the input program involves static analysis, including extracting control flow graphs, call graphs, and identifying implemented security mitigations within the program.
2) Vulnerability detection stage - it uses the symboic execution engine angr to detect the vulnerabilities like many other tools
3) Exploit generation stage - it employs methods like ret to shell, ret to libc, ret to dlresolve. It considers certain cases like for example: If certain gadgets to control the registers it employs a method called ret to csu where it combines the gadgets present in __libc_csu_init.

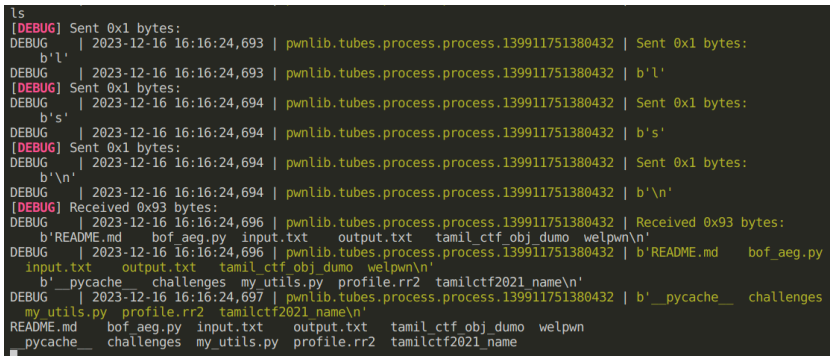
Bof_aeg:
Bof_aeg works as shown in the flowchart below.



(a)

Figure 6: Bofaeg working

Bofaeg utilizes various conditions and operates on each separately. Bofaeg is capable of functioning with binaries featuring Position-Independent Executable (PIE) enabled, provided there is a pre-existing leak in the output. This is particularly effective in solving Capture The Flag (CTF) challenges.
The performance of Bofaeg over Zeratool can be found here - link.
The tool achieves this through a "find_leak" function, which analyzes the output. It then identifies the source of the leak and, if the output is in text format (starting with 0x5555), retrieves the stack base address. Similarly, if it's a libc address (often starting with 7ffff), it obtains the libc base address.



(a)

Figure 7: Binary exploited and finally shell is opened

5.2.1 Limitations:

Both Faeg and Bofaeg utilize the symbolic execution engine angr; however, they overlook the path explosion problem associated with symbolic execution. As a result, detecting stack buffer overflow vulnerabilities in highly complex programs becomes challenging.

Additionally, these tools are not effective when certain security mitigations, such as canary protection, are implemented. In such cases, the canary needs to be leaked through a format string vulnerability and then overwritten onto the buffer when exploiting an overflow vulnerability.

Furthermore, these tools are limited to detecting a single type of vulnerability and are not suitable for identifying complex issues like heap overflows.

6 Future work

Many existing tools may encounter crashes when dealing with large binaries, primarily due to the path explosion problem. Addressing this challenge could involve implementing guided fuzzing, an area that has received considerable attention.

Furthermore, majority of programs primarily focus on detecting overflow and format string vulnerabilities, as they are relatively easier to exploit. However, there is a noticeable gap in research and tools dedicated to identifying more complex vulnerabilities, such as heap overflows.

7 Acknowledgements

I express my gratitude to Prof. Chester Reberio, my guide, for providing guidance throughout the project. I extend my thanks to Mahesh and Aditya Vaichalker for their collaborative efforts and assistance. I would also like to thank the whole Undetectable team for their effective feedback and suggestions during the weekly meets.

8 References

[1] Zeratool, "Github Repository," https://github.com/ChrisTheCoolHut/Zeratool.
[2] Shenglin Xu, Yongjun Wang, "BofAEG: Automated Stack Buffer Overflow Vulnerability Detection and Exploit Generation Based on Symbolic Execution and Dynamic Analysis", Security and Communication Networks, vol. 2022, Article ID 1251987, 9 pages, 2022. https://doi.org/10.1155/2022/1251987=
[3] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," Communications of the ACM, vol. 57, no. 2, pp. 74–84, 2014.
[4] Symbolic execution engine - Angr - https://docs.angr.io/en/latest/
[5] Survey of Automated Vulnerability Detection and Exploit Generation Techniques in Cyber Reasoning System link
[6] Driller : Augmenting Fuzzing Through Selective Symbolic Execution
[7] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2011. Automatic exploit generation. Commun. ACM 57, 2 (2011), 74–84.
[8]Automated Exploitation of Fully Randomized Executables by Austin James Gadient B.S. Computer Engineering United States Air Force Academy (2018)
[9] Binary analysis Radare2
[10] ASLR and pie - https://guyinatuxedo.github.io/5.1-mitigation$_a slr_p ie/index.html$
[11] Format string vulnerability
[12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," Proceedings - IEEE Symposium on Security and Privacy, pp. 380–394, 2012.
[13] LibFuzzer. 2023. LibFuzzer software official website. https://github.com/Dors/libfuzzer-workshop/.
[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 1032–1043.
[15] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu,and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 627–637.

[16] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In 2018 IEEE Symposium on Security and Privacy (SP). IEEE, 697–710.
[17] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E : A Platform for In-Vivo Multi-Path Analysis of Software Systems," Asplos, vol. 46, pp. 1–14, 2011.
[18] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2329–2344.
[19] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2095–2108.
[20] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel.. In NDSS.
[21] Peng Xu, Liangze Yin, Jiantong Ma, Dong Yang, and Wei Dong. 2023. FAEG: Feature-Driven Automatic Exploit Generation. In 14th Asia-Pacific Symposium on Internetware (Internetware 2023), August 04–06, 2023, Hangzhou, China. ACM, New York, NY, USA 9 Pages. FAEG
[22] Bofaeg - Shenglin Xu, Yongjun Wang, "BofAEG: Automated Stack Buffer Overflow Vulnerability Detection and Exploit Generation Based on Symbolic Execution and Dynamic Analysis", Security and Communication Networks, vol. 2022, Article ID 1251987, 9 pages, 2022.
[23] S. Heelan, Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities, Ph.D. thesis. University of, Oxford, 2009.
[24] S. K. Huang, M. H. Huang, P. Y. Huang, C. W. Lai, H. L. Lu, and W. M. Leong, "Crax: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations," in Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability, pp. 78–87, IEEE, Gaithersburg, MD, USA, June 2012.
[25] V. A. Padaryan, V. V. Kaushan, and A. N. Fedotov, "Automated exploit generation for stack buffer overflow vulnerabilities," Programming and Computer Software, vol. 41, no. 6, pp. 373–380, 2015.
[26] L. Xu, W. Jia, W. Dong, and Y. Li, "Automatic exploit generation for buffer overflow vulnerabilities," in Proceedings of the 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 463–468, IEEE, Lisbon, Portugal, July 2018.
[27]D. Repel, J. Kinder, and L. Cavallaro, "Modular synthesis of heapexploits," in Proceedings of the 2017 WorkshoponProgramming Languages and Analysis for Security, pp. 25–35, Dallas, TX, USA, October 2017.
[28] D. Liu, J. Wang, Z. Rong et al., "Pangr: a behavior-based automatic vulnerability detection and exploitation framework," in Proceedings of the 2018 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE), pp. 705–712, IEEE, New York, NY, USA, August 2018.
[29] Y. Wang, C. Zhang, X. Xiang et al., "Revery: from proof-of-concept to exploitable," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1914–1927, Toronto, Canada, October 2018.