SAEG: Stateful Automatic Exploit Generation

Yifan Wu, Yinshuai Li, Hong Zhu, and Yinqian Zhang(✉)

Department of Computer Science and Engineering,
Research Institute of Trustworthy Autonomous Systems,
Southern University of Science and Technology, Shenzhen, China
{11912909,12231139,12012624}@mail.sustech.edu.cn, yinqianz@acm.org

Abstract. The field of Automatic Exploit Generation (AEG) plays a pivotal role in the assessment of software vulnerabilities, automating the analysis for exploit creation. Although AEG systems are instrumental in probing for vulnerabilities, they often lack the capability to contend with defense mechanisms such as vulnerability mitigation, which are commonly deployed in target environments. This shortfall presents significant challenges in exploitation. Additionally, most frameworks are tailored to specific vulnerabilities, rendering their extension a complex process that necessitates in-depth familiarity with their architectures. To overcome these limitations, we introduce the SAEG framework, which streamlines the repetitious aspects of existing exploit templates through a modular and extensible state machine that builds upon the concept of an Exploit Graph. SAEG can methodically filter out impractical exploitation paths by utilizing current information and the target program's state. Additionally, it simplifies the integration of new information leakage methods with minimal overhead and handles multi-step exploitation procedures, including those requiring the leakage of sensitive data. We demonstrate a prototype of SAEG founded on symbolic execution that can simultaneously explore heap and stack vulnerabilities. This prototype can explore and combine leakage and exploitation effectively, generating complete exploits to obtain shell access for binary files across i386 and x86_64 architectures.

Keywords: Automatic Exploit Generation · Symbolic Execution · Vulnerability

1 Introduction

The field of Automatic Exploit Generation (AEG [21]) assists not only in crafting cyber-attacks but also in bolstering cybersecurity defenses. Such tools [28] enable software manufacturers to gauge the severity of their products' vulnerabilities and prioritize their remediation strategies.

In recent times, as vulnerability mitigation strategies become commonplace in modern operating systems, the ability to breach these defense mechanisms

J. Garcia-Alfaro et al. (Eds.): ESORICS 2024, LNCS 14985, pp. 127–145, 2024.
https://doi.org/10.1007/978-3-031-70903-6_7

has emerged as a critical challenge for executing practical attacks in production environments. Numerous vulnerabilities result only in denial of service, failing to compromise sensitive information or facilitate arbitrary code execution, chiefly due to the robustness of security defenses or their inherent limitations. Thus, evaluating vulnerabilities' exploitability under defense mechanisms is a crucial aspect of AEG. For instance, Syzkaller [7] continually reveals a multitude of proof-of-concept (POC) that induce kernel crashes and many of them remain unmitigated yet. However, many POCs do not pose meaningful exploitation threats due to intrinsic vulnerability limitations or robust defense measures. AEG's role is to discern which vulnerabilities can be maneuvered into viable exploits under the constraint of these defense mechanisms.

1.1 Challenges from Modern Protection Mechanisms

Contemporary operating systems and compilers widely support a quartet of distinguished defense mechanisms: Data Execution Prevention (DEP), often known as the NX bit, Stack Canaries, Address Space Layout Randomization (ASLR), and RELocation Read-Only (RELRO). DEP is designed to deter the execution of code from data pages, effectively preventing certain types of exploits such as shellcode injection. Stack Canaries safeguard the call stack with a secret value to detect buffer overflow attempts, while ASLR systematically randomizes memory addresses to hinder attackers from predicting target locations. The RELRO feature marks the Global Offset Table (GOT) as read-only after initialization to prevent attackers from hijacking the control flow by tampering with the GOT. Consequently, in scenarios where these protective measures are enabled, attackers typically need to adeptly combine multiple techniques to carry out attacks through information leakage and bypass these protections.

Table 1. Protections bypass ability of recent AEG framework

Framework	NX	Canary	ASLR
Zeratool [18]	◐	○	●
BOF AEG [27]	●	○	●
LAEG [14]	◐	◐	◐
PANGR [12]	●	○	○
ExpGen [9]	●	○	◐
CanaryExp [10]	●	◐	◐
SAEG	●	●	●

● means that the framework can bypass this defense with several techniques
◐ means that the framework can bypass this defense with one technique
○ means that the framework can not bypass this defense

After evaluating existing implementations, we noticed that most frameworks neglect complex information leakage challenges. As indicated in Table 1, few efforts have been made to concurrently address NX, ASLR, and Canary. All current open-source frameworks fall short in circumventing Canary and other CFI checks. However, these three mechanisms are widely supported by modern compilers such as gcc and clang and are extensively used in operating systems like OpenBSD 7.4, where the default clang-local compiler ships with all three enabled. Thus, given the lack of general capability among present AEG frameworks to manage such complex exploitation efforts, the generation of exploits involving complex information leaks represents a meaningful yet challenging endeavor.

1.2 Our Solutions

Complex exploits typically involve a multistep process to progressively gather new information and ultimately complete the attack. Consequently, there is a need for a fine-grained exploit system that can efficiently manage and filter the necessary steps. Towards this end, we have crafted an expandable exploitation graph based on attack graphs [11] to accurately depict each step in the attack sequence and label the acquired information. We have developed an algorithm to generate these exploitation graphs using the target program and primitive attack templates. The graph's analysis is carried out through symbolic execution to find potential successor nodes. If the exploit generation is successful, we will obtain a complete path composed of nodes in the exploitation graph.

Here, we list the main contributions of our work:

- (1) We have developed and implemented SAEG, an innovative AEG framework that utilizes an exploitation graph derived from attack graphs to accurately manage the steps involved in exploiting vulnerabilities.
- (2) We introduce a refined approach to AEG that utilizes primitive attack templates. In contrast to traditional AEG solutions dependent on complete attack templates, our method simplifies the generation of complex exploits capable of bypassing modern protection measures and improves extensibility.
- (3) Our evaluation of 34 real Capture The Flag (CTF) challenges indicates that SAEG can produce intricate exploits involving multiple steps. Compared to contemporary frameworks, SAEG's exploit generation efficiency is 5.4x greater on average, and it can generate exploits beyond the capabilities of those frameworks.
- (4) We release the source code of SAEG and the test cases used in the experiments at https://github.com/GhostFrankWu/SAEG.

2 Background

The concept of attack graphs was initially conceptualized to represent the entirety of a cyber-attack process, encompassing stages such as initial access,

privilege escalation, and lateral movement. Analogously, in the context of binary program exploitation, crafting an interactive shell exploit often necessitates the orchestration of various vulnerability exploitation techniques. To streamline the automation of complex exploits requiring multiple exploitation stages, we introduce the exploitation graph, an evolution of attack graphs designed specifically for binary program attack sequences.

Exploitation Techniques

Traditional binary exploitation is mainly categorized by the target data region: stack or heap. Exploitation based on the stack primarily manipulates the stack buffer overflow for control flow hijacking. It employs return-oriented programming (ROP) to counteract DEP protections and leverages format string vulnerabilities to facilitate information leakage, thereby subverting Canary and ASLR defenses. Exploitation based on the heap primarily involves constructing a desired heap layout, often referred to as heap Feng Shui, and leveraging vulnerabilities such as heap overflows, use-after-free, and double-free to enable information leakage and arbitrary address writes for exploitation.

Stack-Based Buffer Overflow. One typical scenario of Stack-based Buffer overflow [13] is a function like *strcpy* neglects to enforce buffer boundaries, there is a risk of overwriting adjacent memory, potentially resulting in the corruption of neighboring stack frame elements, including the Canary, base pointer, and return address. These vulnerabilities can lead to shellcode injection or ROP attacks, enabling unauthorized attackers to obtain the entire shell access.

An AEG framework needs the capability to perceive the program's runtime state, plan paths, and correctly overwrite the Canary and return address after achieving information leakage, ultimately hijacking the program's control flow upon function return.

Format String Vulnerability. Exploiting format string vulnerabilities is a common technique for information leakage and sometimes also a way to hijack control flow. When the $printf()$ function's first parameter (the format string) is under user control, attackers can carefully construct special format specifiers like $\%s$, $\%5\$p$, $\%hn$, etc., to crash the program or leak data from the stack, arbitrary addresses, or achieve arbitrary address writes.

The **va_list** pointer in $printf()$ sequentially reads data from the stack. At this point, an attacker can modify $[num]$ using a format like $\%[num]\$[fmt]$, specifying the offset relative to the initial position of **va_list**. By using $[fmt]$, the attacker controls the behavior of the $printf()$ function. Attackers can access controllable memory using specific offsets and achieve arbitrary memory read/write using indirect addressing format specifiers like s and n.

Depending on the length and content restrictions of controllable characters in the vulnerability's environment and the varying ability to manipulate data on the stack, the exploitation capability of format string vulnerabilities also

differs. The ability to achieve information leakage through format string attacks typically has broad requirements. Therefore, implementing automatic detection and exploitation of format string vulnerabilities can enhance the framework's ability to leak information and provide the chance of control flow hijacking by locating and overwriting function pointers.

Return Oriented Programming. Return Oriented Programming (ROP) is a commonly employed technique designed to counteract the Non-Executable (NX) protection of data segments [20]. ROP involves the identification of usable code snippets, often referred to as gadgets, within code segments marked as executable but not writable through static analysis. These gadgets typically end with a *ret* instruction and are frequently employed for register manipulation and stack data read/write operations. Attackers can construct various powerful primitives by strategically placing consecutive code addresses on the stack through vulnerabilities such as buffer overflows. This enables them to achieve function parameter layout and invoke functions at arbitrary addresses.

ROP plays a pivotal role in multiple exploitation stages, including information leakage. Therefore, nested ROP scenarios are often encountered in a comprehensive exploitation. For instance, in cases where an attacker has limited access to imported functions within the program, they may initiate a call to $puts(\&puts)$; to obtain the address of the puts function in libc. After returning to the stack overflow point, they can trigger the overflow again and call any function, such as $exit(0)$;, located within libc.

Leveraging leaked random address information, it is possible to combine functions and code snippets from the dynamically loaded library for a function call similar to $system(\text{“}/bin/sh\text{”})$;. Alternatively, in scenarios where DEP is disabled, it becomes feasible to inject shellcode into controllable regions and execute arbitrary code. In many early AEG approaches, extensive research was conducted on automatic shellcode generation and injection. However, the main focus of this work is to address complex challenges such as information leakage in automated exploitation. Therefore, the injection of shellcode is considered only as a means of introducing a scalable and simplified state for us.

Nested ROP presents diverse combinations of function call chains, demanding that an AEG framework selectively and flexibly choose and combine these calls. This step-by-step approach is essential for overcoming various protections and ultimately gaining the ability to execute arbitrary code.

3 Design

3.1 Methodology

Our framework SAEG can be conceptualized as navigating an exploitation graph (EG) through a specific search strategy to probe potential exploit paths. The traversal of the EG is guided by conditional transitions and prioritizations that limit the breadth of the state to be explored. The EG is defined as follows:

Definition 1 (Exploit Graph). Let AP be a set of atomic information such as secret canary value or randomized base address of `.text` segment. An exploit graph or EG is a tuple $EG = (S, \tau, S_0, S_s, L)$, where S is a set of states, $\tau \subseteq S \times S$ is a transition relation, $S_0 \subseteq S$ is a set of initial states, $S_s \subseteq S$ is a set of success states, and $L : S \rightarrow 2^{AP}$is the labeling of states with a set of information true in that state. Intuitively, S_s denotes exploitation completed, for example, reaching a shell access.

Within the Exploit Graph (EG), each state corresponds to the exploitation of a vulnerability with an information set. Such vulnerabilities encompass stack overflows, format string vulnerabilities, and use-after-free, among others. We maintain the distinctness of each state by associating it with a specific type of vulnerability and its present information set where $s_i.l = \mathrm{L}(s_i)$. Every $\tau_i \subseteq \tau$ symbolizes an individual atomic exploit method.

To construct the EG, we first manually create an attack template library (T) consisting of several atomic exploitation steps. Each attack template contains the classification of the techniques to which the vulnerability is exploited, the premises required to execute this exploitation, the atomic information obtained from completing the exploitation, and the code knowledge to execute the attack. The definition of the attack template is as follows:

Definition 2 (Attack Template). An attack template is a tuple $t = (type, p, g, \mathrm{code})$ where:

- *type* represents the vulnerability type exploited by the template.
- $p \subseteq 2^{AP}$ provides the preconditions of the template.
- $g \in AP$ is a subset of atomic information, denoting the effect of the attack.
- code is the payload of the exploitation.

SAEG generates the initial states S_0 based on the input binary program. For instance, if the target binary is not compiled with Position Independent Executable (PIE), each initial state $s \in S_0$ implicitly contains the base address of the `.text` segment. Consequently, the generated EG excludes techniques aimed at revealing this known information. Similarly, if the target binary has not disabled lazy binding, the EG would include exploits such as GOT hijacking. Following this, SAEG leverages the predefined attack templates in conjunction with S_0 to construct the EG, as detailed in Algorithm 1.

Subsequently, SAEG constructs an exploit path by recursively traversing the program's execution tree and searching for new vulnerabilities by symbolic execution. The transfer from the current state to its successor indicates the deployment of an attack template according to the newly found vulnerability. SAEG also verifies the exploit path during its construction and returns a complete exploit path if found otherwise it returns empty if all available exploits are failed. We define that SAEG verifies the exploit through the CHECK_EXPLOITATION procedure. The primary concern of this verification procedure is ensuring that the exploit's payload conforms to the target program's constraints. For example, if an attack requires overwriting 16 bytes of data, but only overwrite 12 bytes can

be overwritten under the current state, then it does not meet the constraint. The process through which SAEG generates exploits based on the EG is shown in Algorithm 2.

Algorithm 1. Generate EG

Require: A set of attack templates T, initial states S_0
Ensure: EG $= (S, \tau, S_0, S_s, L)$

```
1: function GEN_EG(T, S_0)
2:     S ← ∅, S_s ← ∅, L ← ∅, τ ← ∅, W ← S_0
3:     while W ≠ ∅ do
4:         s ← pop(W)
5:         S ← S ∪ {s}
6:         if shell_access ∈ L[s] then
7:             S_s ← S_s ∪ {s}
8:         else
9:             for all t ∈ T do
10:                if t.g ∈ L[s] then
11:                    continue
12:                end if
13:                if t.p ∈ L[s] then
14:                    Create a new state s'
15:                    s'.l ← L[s] ∪ t.g
16:                    L[s'] ← s'.l
17:                    τ[s, s'] ← t
18:                    W ← W ∪ {s'}
19:                end if
20:            end for
21:        end if
22:    end while
23:    return EG = (S, τ, S_0, S_s, L)
24: end function
```

Algorithm 1 is capable of ensuring each step of the vulnerability exploitation will acquire new information, this provides a constraint implied at line 8 of Algorithm 2 to limit the recursion depth to less than $|AP|$. As a result, Algorithm 2 is inherently protected against the problem of state explosion, yet the nature of symbolic execution itself may still lead to generating a large number of states.

Overall, the work of SAEG is divided into two steps: (1) Generate EG employing Algorithm 1 with attack templates and the target program. (2) Try to generate exploits by analyzing the program against the EG with Algorithm 2.

Algorithm 2. Generate exploitation from EG

Require: An exploit graph EG, initial states S_0, a sequence of transitions τ_a
Ensure: An exploit path consisting of a sequence of transitions if exists, otherwise $\emptyset$

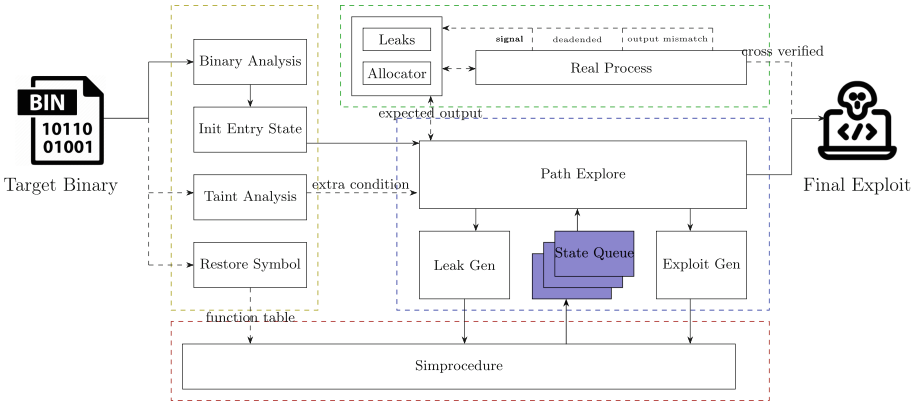
```
1: function CHECK_EG(EG, sn, τ_a)          ▷ We use symbolic execution to get all immediate successor of sn
2:     for all transition τ[s, s'] in EG.τ where s' is an immediate successor of sn do
3:         τ'_a ← τ_a ∪ {τ[s, s']}          ▷ Append τ[s, s'] to τ_a to create a new path
4:         if CHECK_EXPLOITATION(τ'_a) then
5:             if s' is in EG.S_s then
6:                 return τ'_a
7:             else
8:                 result ← CHECK_EG(EG, s', τ'_a)
9:                 if result ≠ ∅ then
10:                    return result                ▷ Found An exploit path
11:                end if
12:            end if
13:        end if
14:    end for
15:    return ∅                                     ▷ No exploit path found
16: end function
```

3.2 Architecture

We present a prototype of SAEG as shown in Fig. 1. The entire prototype is primarily divided into four major modules: binary preprocessing, exploit graph verification, Hook module, and verification module.

The solid arrows in the figure denote state transitions and the dotted arrows indicate information transfer. The binary preprocessing and exploit graph verification aim to extract the initial states and execute Algorithms 1 and 2. The EG verification begins by taking the initial state obtained through preprocessing and conducting symbolic execution to traverse the execution tree. It aims to generate and verify multiple immediate successors for EG in scenarios such as memory reading and writing, hook hits, and function returns. To ensure consistency, the path verified by EG is synchronously passed to the interactive framework for verification within the actual running program.

The Hook framework provides an appropriate abstraction for library functions. On one hand, hooking library functions can alleviate the issue of state explosion that occurs during the symbolic execution of these functions, thereby significantly enhancing the efficiency of symbolic execution and even directly affecting its exploitability. On the other hand, the hook framework allows the overall framework to be aware of the program's state. This enables the framework to infer the controllability of the content and length of library function arguments. It facilitates the exploitation of library functions such as *printf*



Four major modules: binary preprocessing (depicted in yellow), exploit graph verification (depicted in blue), hook module (depicted in red), interaction verification module (depicted in green)

Fig. 1. Schematic representation of the implementation structure of SAEG. (Color figure online)

and *puts*, and also aids in identifying the offsets of sensitive data, such as stack addresses and canaries, thereby enabling effective and accurate information leakage.

In contrast to traditional open-source AEG frameworks like BOF AEG and Zeratool, expanding our framework only requires extending specific edges from attack templates. This eliminates the need for writing multiple sets of modules for designing from discovering exploitation states to exploitation models. SAEG efficiently supports the combination of various information leakage and exploitation methods with concise code. Such capability is challenging to achieve in traditional expert systems based on greedy implementations that can only prioritize a single goal.

Our approach leverages strategies such as prioritization and pruning to enhance exploit attempts. In a scenario where multiple exploitable vulnerabilities converge within line 2 of Algorithm 2, the SAEG framework is adept at sequencing attempts according to a set of manually defined priorities. These priorities may include the likelihood of successful exploitation or the risk of causing program crashes. Furthermore, we have refined the preprocessing phase, enabling the framework to discard more extraneous paths contingent on the preprocessing outcomes. Such optimization has led to substantial performance enhancements as evidenced by practical trials. By utilizing the exploit graph, SAEG can orchestrate a variety of exploit techniques and effectively bypass security mechanisms such as NX, ASLR, and Canary.

3.3 Example

To briefly demonstrate the workflow of SAEG, We use the vulnerable program in Fig. 2 compiled with both Canary and NX enabled, but PIE disabled.

```
1 int main(){
2     char buf[8];
3     read(stdin, buf, 80);   // overflow
4     printf(buf);            // data leak
5     read(stdin, buf, 80);   // overflow
6 }
```

Fig. 2. Simple vulnerable Program

SAEG first analyzes the protection status of the program and generates $S_0 = \{start\}$. For this simple scenario, we provide four primitive templates for vulnerability exploitation $T = \{A_1, A_2, ..., A_4\}$, which correspond to *Attack* 1 to *Attack* 4 in the appendix. Subsequently, the EG_{basic} generated by SAEG, $\text{EG}_{basic} = \text{Gen\_EG}(T, S_0)$, is shown in Fig. 3.

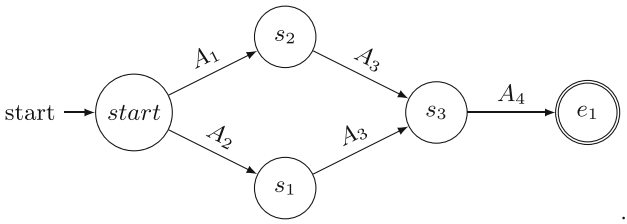


Fig. 3. Generated EG_{basic}

Subsequently, starting from *start*, SAEG performs symbolic execution to line 4 of the source code, where two potential vulnerabilities s_1 and s_2 emerge. Assuming SAEG prioritizes the attack format-string-leak-canary (A_2), it will first verify whether A_2 can be completed. Once the attack is confirmed to be feasible, SAEG recursively continues symbolic execution from s_1 until it finds s_3 and verifies that A_3 can be completed. If at this point A_3 is considered not satisfying the constraints, SAEG will backtrack to *start* to re-verify whether A_1 can be completed. This process continues until a transition to the termination state $e_1 \in S_s$ is made, at which point the entire exploitation path constitutes the complete exploit generated. If, upon backtracking to *start*, there are no new potential exploits, then the exploit generation fails.

For a slightly more complex situation, the program is compiled with both the PIE and the canary enabled. Now the attacker also needs to obtain the address of the `.text` segment randomized by ASLR. In the new initial state, $L(start') = \emptyset$. We added two attack templates to the template library, $T' = \{A_1, ..., A_6\}$, corresponding to *Attack* 1 to *Attack* 6 in the appendix. The $\text{EG}_{enhanced}$ generated by SAEG now is as shown in Fig. 4.

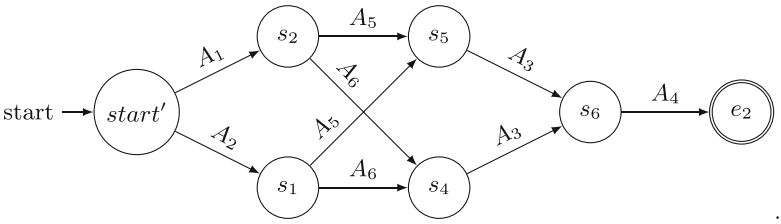


Fig. 4. Generated $\text{EG}_{enhanced}$

Then when the SAEG inspection of *Attack* 1 can be completed, it will continue to try to leak more information, such as by appending the formatted string length to complete *Attack* 5, or continue to exploit the formatted string attack to complete *Attack* 6. Thus, by adding two templates, we have expanded SAEG to complete the complex exploit generation with more information leakage.

4 Implementation

In the detailed implementation, the binary preprocessing module includes the Radare2 [17] analysis engine and FLIRT [8] function signature recognition algorithm. Radare2 [17] is used for the reverse analysis of binary files, detecting high-risk functions and code snippets. FLIRT [8] identifies statically compiled library functions to alleviate the burden of symbolic execution. exploit graph verification module uses angr [22] for symbolic execution, which uses z3 solver [5] as its backend. Exploits satisfying the constraint conditions are interactively tested with the local executable file or remote port service by the verification module. The prototype of SAEG is implemented in Python using 2240 lines of code, which includes 30 attack templates. Among them, four templates exemplify heap exploitation by implementing the house-of-force attack [25]. On average, each attack template requires fewer than 50 lines of Python code, making the prototype relatively easy to further expand.

5 Evaluation

In this section, we evaluate the effectiveness of SAEG through binary files from several CTF challenges. The assessment primarily focuses on protection strategies, exploitable vulnerability types, and the performance of generating exploit

samples. All experiments without "*" mark were conducted on a Ubuntu 20.04 in docker with 2 Intel i7-13700 cores under virtualization of QEMU Virtual CPU version 2.5+ @ 2.1 GHz and 4 GB memory. It should be noted that since we did not have access to the source code of LAEG, the results of LAEG were referenced from the original LAEG work on an Ubuntu 18.04.5 LTS with 6 AMD R5 R5600X CPU @ 3.7 GHz cores and 16GB memory. Despite having fewer CPU cores, less memory, and a lower clock frequency, SAEG still exhibits higher efficiency. The details are shown in Table 2.

Compared to existing relevant work, our approach achieves a significant efficiency improvement without sacrificing the capability of exploitation. We posit that the efficiency enhancement primarily stems from the more effective pruning of conditional branches by the exploit graph, leading to a notable increase in vulnerability exploitation efficiency. Note that many relevant implementations rely on complete template matching, requiring significant time for matching and verifying each template. While SAEG generates numerous exploit paths by combining primitive templates, it efficiently minimizes unnecessary verification guided by the exploitation graph. Simultaneously, the exploit graph proves effective in handling complex exploit generation, as it can adeptly manage multi-step states, thereby bypassing defenses in binary programs equipped with diverse protective mechanisms. We further validate the feasibility of our framework in addressing heap exploitation challenges.

In the two samples from defcon27_speedrun, SAEG demonstrated a significant advantage over the top hackers participating in DEF CON. The participants exploited the vulnerabilities in 266 s and 280 s respectively, while SAEG outperformed the fastest participant by 35 times and 37 times, confirming its efficiency in rapidly identifying exploitable vulnerabilities. This underscores the value of AEG in compelling assessments of software vulnerability remediation, namely, having a greater chance of confirming the urgency of fixes before live exploitation, while also allowing time for remediation efforts, post the public disclosure of POCs.

In the defcon27_speedrun-002 example, our approach's runtime is slightly close to that of LAEG, attributed to the presence of numerous constraint-solving steps in the file (requiring interactive password input). LAEG provides a crash POC through the exploitation framework, including the correct password, resulting in saved solving time. Given that the real exploitation process may significantly differ from the POC's execution flow (for instance, in a repeatable path where one branch's two sides represent a crash and information leakage, the POC may only obtain crash information, neglecting the information leakage on the other side of the branch-essential for completing the exploitation), we contend that the proper handling and utilization of POC information are worthwhile topics for exploration. However, this exceeds the scope of our current work.

In testing scenarios such as static compilation examples (S-crash-static), the input target programs are all large binary files exceeding 500KB. Only SAEG and LAEG can generate the exploitation, and the efficiency of SAEG is higher

Table 2. Evaluation results of SAEG (time: seconds)

Vulnerable Program	Protection	Zeratool	BOF AEG	LAEG	SAEG
umdctf_jne	N	7 (9.33x)	2.69 (3.58x)	-	0.75
downunderctf_out	N	6 (7.06x)	✗	-	0.85
csawctf_roppity	N	30 (16.2x)	2.38 (1.29x)	-	1.85
downunderctf_return	N	30 (30.0x)	2.39 (2.39x)	-	1.00
dctf_babybof	N	27 (24.3x)	1.59 (1.43x)	-	1.11
umdctf_jnw	N	27 (26.5x)	2.39 (2.34x)	-	1.02
csictf_0x1	N	✗	2.40 (2.72x)	-	0.88
csictf_0x2	N	✗	2.40 (9.99x)	-	0.24
csictf_0x3	N	7 (8.75x)	2.36 (2.95x)	-	0.80
dctf_sanity	N	✗	10.7 (12.4x)	-	0.86
csawctf_password	N	✗	60 (7.22x)	-	8.31
hcktivityctf-retcheck	R+N	✗	3.09 (3.90x)	-	0.79
tamilctf_name	N	✗	1.63 (1.81x)	-	0.90
dicectf_babyrop	N	✗	2.32 (2.19x)	-	1.06
utctf_resolve	N	✗	2.35 (2.17x)	-	1.08
nahamconctf smol	N	✗	1.56 (1.88x)	-	0.83
sharkctf_give	R+A+N	✗	4.00 (4.00x)	-	1.00
angstromctf_no_canary	N	✗	6.40 (3.65x)	*8.60 (4.90x)	1.75
angstromctf_tranquil	N	✗	3.80 (1.70x)	*5.59 (2.51x)	2.23
wpictf_dorsia1	A+N	✗	✗	-	1.28
downundercrf_deadcode	N	✗	✗	-	0.29
redpwncrf_coffer	N	✗	✗	-	0.28
dctf_hotelrop	A+N	✗	✗	-	1.11
lexingtoncrf_gets	A+N	✗	✗	-	0.69
crash_backdoor	-	10 (9.52x)	1.75 (1.72x)	*5.72 (5.56x)	1.02
crash_canary	C	✗	✗	*5.60 (4.88x)	1.14
crash_pie	A	✗	✗	*5.74 (4.85x)	1.18
S-crash-static	-	✗	✗	*1.68 (1.73x)	0.96
S-defcon27-speedrun-001	N	✗	✗	*41.3 (5.46x)	7.55
defcon27-speedrun-002	N	✗	✗	*6.20 (1.54x)	4.02
utctf_bof	N	✗	2.33 (1.60x)	*5.54 (3.81x)	1.45
gyctf_force	R+C+A+N	✗	✗	-	174
lexingtoncrf-madlibs	R+N	✗	✗	-	✗
crash-canary-pie	R+C+N	✗	✗	✗	✗

The number in parentheses after the time refers to the efficiency improvement multiple achieved by our SAEG compared to the current framework. In the protection status, "R" represents RELRO, "C" represents Canary, "A" represents ASLR, and "N" represents NX. The "S-" prefix indicates a statically linked program.

than that of LAEG with existing POC knowledge. This indicates that modeling library functions and attempting to reconstruct function tables is beneficial.

The example lexingtonctf_madlibs is a typical example of the limitation of symbolic execution. This example contains a nested format string vulnerability, which can lead to a stack overflow vulnerability. However, symbolic execution based on angr still cannot handle this problem. Therefore, SAEG is unable to complete the exploitation as it cannot find new potential vulnerabilities. Another failed example is crash-canary-pie. SAEG has the capability to handle multi-step leaks, but this scenario needs to involve truncation symbol overwrite vulnerability. In order to exploit this vulnerability, attackers may need to overwrite the low bytes of the return address pointing to libc to re-enter the current function, creating a second chance for information leaks and exploiting stack overflow vulnerability. SAEG does not have an exploitation primitive template for re-entering the current function, thus it cannot complete the full exploitation. After we provided a simple re-entry template, SAEG successfully exploited crash-canary-pie. However, this indicates that SAEG needs to combine existing templates rather than create new exploitation primitives beyond the templates.

6 Discussion

Real-World Applications

SAEG may encounter inherent limitations of symbolic execution in collecting vulnerability states, such as path explosion issues in complex large-scale applications. In such cases, we can replace the path exploration module with known exploit samples, such as POC causing a denial of service, to improve the efficiency of initial exploitation state exploration for large-scale applications. Alternatively, we can use concolic execution to enhance efficiency through fuzzing. While recent works, such as MAZE and Revery, focus on optimizing modules or smaller components of exploit generation, the topic of crash state exploration, as another field of program analysis, is not extensively discussed in this article.

SAEG's exploit generation and exploitation information collection capabilities can help developers differentiate between vulnerabilities that may only lead to DoS attacks and those that have the potential to achieve arbitrary code execution by bypassing protection mechanisms. By analyzing the potential harm of these vulnerabilities and understanding the underlying causes of successful exploitation, SAEG can assist developers in proposing more secure patching solutions, ultimately improving the overall security of modern protections.

Trade-Off of Symbolic Execution

Using modeling and hooking can mitigate the problem of state explosion caused by library functions, but it may cause the symbolic execution engine to lose precise information about stack data. Since the stack space is linear and not automatically cleared, both library functions and the program itself reuse the

same stack memory. Therefore, crucial information introduced by library functions may reside on the stack and it is difficult for symbolic execution to capture after function replacement. This information may be obtainable through concolic execution or real-time debugging methods. However, using dynamic analysis to obtain this part of information leakage is beyond the scope of our framework.

Future Research

The current limitations of SAEG are primarily associated with a clear reliance on symbolic execution and ROP technology. ROP technology cannot be applied to high-risk vulnerabilities in real scenarios caused by logical flaws, such as command concatenation [16], branch mispredictions, and supply chain backdoors [15], where attacks do not manipulate return addresses/jump addresses. To address these limitations, future work can focus on optimizing the symbolic execution section to handle high-risk path exploration and make the framework compatible with these types of vulnerability exploitation.

7 Related Works

7.1 AEG

The research related to AEG was first proposed in APEG [3], which aims to generate exploits for unpatched programs through a patched program. In subsequent work [1], AEG automatically generates an exploit by providing a POC. In recent years, researchers have proposed several AEG solutions. However, modern protection procedure like ASLR and Stack Canary brings great challenges to the exploitation capabilities of those AEG implementations.

Exploit Generation. Many AEG frameworks strive to efficiently generate utilization like MAZE [24] tackles heap layout as a patching problem by identifying allocated and deallocated primitives to construct the desired heap layout in complex scenarios. Revery [23] efficiently explores and generates exploits by replacing symbolic execution with fuzzing. However, they assume protections like ASLR that require information leakage are disabled. Therefore, combining SAEG with these technologies may improve the exploitation ability of existing work.

Protection Bypass. The realm of automatic exploit generation stands as a vibrant domain in contemporary research endeavors, some recent work has focused on solving the problem of information leakage in AEG. For example, LAEG [14] aims to make up for the lack of information leakage capabilities of existing AEGs through optimized dynamic analysis methods. ExpGen [9] aims to obtain leaked ASLR information through fuzz testing and make subsequent use of it, while CanaryExp [10] adds detection of Canary leaks similar to LAEG on the basis of ExpGen, and enhances the leakage capabilities by guiding fuzz

testing technology while making the leaks more Robust. However, it is difficult for them to use formatted strings or heaps of feng shui to accomplish more complex information leakage or vulnerability exploitation. Autopwn [26] implements the first Artifact-assisted automatic heap exploitation framework, which defines an Exploitation State Machine to receive automatically learned knowledge to efficiently perform complex heap exploitation.

7.2 Path Exploration

Symbolic Execution. Symbolic execution utilizes symbols to represent variables and simulate the execution of a program. It extracts and resolves constraints along the execution path using constraint solvers like z3 and cvc5 [2]. Theoretically, symbolic execution generates constraints for every path, which may result in challenges such as state explosion.

Prominent tools for symbolic execution include KLEE [4] and angr. KLEE functions as a source code analysis framework, compiling from the source code and incorporating instrumentation for symbolic execution. On the other hand, angr serves as a platform-agnostic binary analysis framework, offering a plethora of easily extendable modules and interfaces. It can simulate binary program instructions and convert standard or file inputs into bit vectors.

Fuzzing. Fuzzing can explore a large number of paths in a short period of time through processes such as the automated generation and mutation of inputs. Common tools for fuzzing include American Fuzzy Lop (AFL) [30] and AFL++ [6]. AFL utilizes genetic algorithms to effectively expand code coverage, and it can be further improved through the integration of ASAN [19] and QEMU.

For example, ARCHEAP [29] uses extended AFL to automatically detect new primitives for heap exploitation and CanaryExp uses AFL++ to generate POC that leaks the value of canary.

8 Conclusion

We have designed a novel exploitation graph based on the attack graph. Additionally, we have implemented an automatic exploit generation framework, SAEG, based on the exploitation graph. This framework relies on finely-grained vulnerability primitives that are easy to expand and can generate complex exploits containing information disclosure to counter general modern protection mechanisms. We use binary files from CTF challenges to evaluate the framework, and the results demonstrate the effectiveness of SAEG. Lastly, we believe we have raised the state-of-art of open-source AEG frameworks and provided new ideas for the state management of AEG.

Appendix for Attack Templates

Attack 1 chop-leak-canary

Type: Overwrite terminate symbol in output string.
Premise: None.
Effects: Secret value of canary.

Attack 2 format-string-leak-canary

Type: use stack format string vulnerability to leak data on stack.
Premise: None.
Effects: Secret value of canary.

Attack 3 leak-got

Type: Use stack overflow to launch ROP attack.
Premise: Secret value of canary $\wedge$ Base address of `.text` segment.
Effects: Base address of `.libc` mapping.

Attack 4 return-to-libc

Type: Use stack overflow to launch ROP attack.
Premise: Secret value of canary $\wedge$ Base address of `.libc` segment.
Effects: Shell access.

Attack 5 chop-leak-text

Type: Overwrite terminate symbol in output string.
Premise: None.
Effects: Base address of `.text` segment.

Attack 6 format-string-leak-text

Type: use stack format string vulnerability to leak data on stack.
Premise: None.
Effects: Base address of `.text` segment.

References

1. Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D.: Automatic exploit generation. Commun. ACM **57**(2), 74–84 (2014)
2. Barbosa, H., et al.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24

3. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: techniques and implications. In: 2008 IEEE Symposium on Security and Privacy (SP 2008), pp. 143–157. IEEE (2008)
4. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol. 8, pp. 209–224 (2008)
5. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
6. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: {AFL++}: Combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20) (2020)
7. Google: Syzkaller: an unsupervised coverage-guided kernel fuzzer (2006). github.com/google/syzkaller. Accessed 22 Dec 2023
8. Guilfanov, I.: F.l.i.r.t: Fast library identification and recognition technology (1997). hex-rays.com/products/ida/tech/flirt. Accessed 22 Dec 2023
9. Huang, H., Lu, Y., Pan, Z., Zhu, K., Yu, L., Zhang, L.: ExpGen: a 2-step vulnerability exploitability evaluation solution for binary programs under ASLR environment. Appl. Sci. **12**(13), 6593 (2022)
10. Huang, H., Lu, Y., Zhu, K., Zhao, J.: CanaryExp: a canary-sensitive automatic exploitability evaluation solution for vulnerabilities in binary programs. Appl. Sci. **13**(23), 12556 (2023)
11. Jha, S., Sheyner, O., Wing, J.: Two formal analyses of attack graphs. In: Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15, pp. 49–63. IEEE (2002)
12. Liu, D., et al.: PanGR: a behavior-based automatic vulnerability detection and exploitation framework. In: 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 (2018)
13. MITRE: Cve-cwe-121: Stack-based buffer overflow (2006). cwe.mitre.org/data/definitions/121. Accessed 22 Dec 2023
14. Mow, W.L., Huang, S.K., Hsiao, H.C.: Laeg: leak-based AEG using dynamic binary analysis to defeat ASLR. In: 2022 IEEE Conference on Dependable and Secure Computing (DSC), pp. 1–8. IEEE (2022)
15. NIST: Realtek jungle SDK remote code execution vulnerability (2021). nvd.nist.gov/vuln/detail/cve-2021-35394. Accessed 22 Dec 2023
16. NIST: Apache spark command injection vulnerability (2022). nvd.nist.gov/vuln/detail/cve-2022-33891. Accessed 22 Dec 2023
17. radareorg: Radare2 (2023). github.com/radareorg/radare2. Accessed 22 Dec 2023
18. Roberts, C.: Zeratool: automatic exploit generation (AEG) and remote flag capture for exploitable CTF problems (2023). https://github.com/ChrisTheCoolHut/Zeratool. Accessed 22 Dec 2023
19. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: {AddressSanitizer}: A fast address sanity checker. In: 2012 USENIX annual technical conference (USENIX ATC 12), pp. 309–318 (2012)
20. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 552–561 (2007)
21. Thanassis, H.A., Kil, C.S., David, B.: AEG: automatic exploit generation. In: Ser. Network and Distributed System Security Symposium (2011)

22. Wang, F., Shoshitaishvili, Y.: ANGR - the next generation of binary analysis. In: 2017 IEEE Cybersecurity Development (SecDev), pp. 8–9. IEEE (2017)
23. Wang, Y., 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 (2018)
24. Wang, Y., Zhang, C., Zhao, Z., Zhang, B., Gong, X., Zou, W.: {MAZE}: towards automated heap Feng Shui. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 1647–1664 (2021)
25. Xie, T., Zhang, Y., Li, J., Liu, H., Gu, D.: New exploit methods against ptmalloc of glibc. In: 2016 IEEE Trustcom/BigDataSE/ISPA, pp. 646–653. IEEE (2016)
26. Xu, D., Chen, K., Lin, M., Lin, C., Wang, X.: AutoPWN: artifact-assisted heap exploit generation for CTF PWN competitions. IEEE Trans. Inf. Forensics Secur. (2023)
27. Xu, S., Wang, Y.: Bofaeg: automated stack buffer overflow vulnerability detection and exploit generation based on symbolic execution and dynamic analysis. Secur. Commun. Netw. **2022** (2022)
28. Yang, S., Chen, K., Wang, Z., Zhang, C.: Exploit-oriented automated information leakage. Int. J. Sustain. Dev. Plan. **17**(5) (2022)
29. Yun, I., Kapil, D., Kim, T.: Automatic techniques to systematically discover new heap exploitation primitives. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 1111–1128 (2020)
30. Zalewski, M.: American fuzzy lop (2006). lcamtuf.coredump.cx/afl. Accessed 22 Dec 2023