applied sciences

MDPI

Article

ExpGen: A 2-Step Vulnerability Exploitability Evaluation Solution for Binary Programs under ASLR Environment

Hui Huang, Yuliang Lu *, Zulie Pan, Kailong Zhu, Lu Yu and Liqun Zhang

College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China; huanghui17@nudt.edu.cn (H.H.); panzulie17@nudt.edu.cn (Z.P.); zhukailong@nudt.edu.cn (K.Z.); yulu@nudt.edu.cn (L.Y.); zhanglq@nudt.edu.cn (L.Z.)

* Correspondence: luyuliang@nudt.edu.cn

Abstract: Current automatic exploit generation solutions generally adopt an 1-step exploit generation philosophy and neglect the potential difference between analysis-time environment and runtime environment. Therefore, they usually fail in evaluating exploitability for vulnerable programs running in an ASLR environment. We propose ExpGen, a 2-step vulnerability-exploitability evaluation solution for binary programs running in an ASLR environment, with three novel techniques introduced, separately partial-exploit sensitive-POC generation, exploitation context sensitive analysis-time exploit generation, and runtime exploit relocation. ExpGen firstly generates an analysis-time exploit that can carry out all the desired exploitation steps through applying the first two techniques in an iterative manner, then dynamically gaps the address-space layout differences between the analysis-time environment and runtime environment by adopting the runtime exploit-relocation technique, making the analysis-time exploit dynamically adaptable to the runtime exploitation session. Using a benchmark containing six test programs, 10 CTF&RHG programs and four real-world applications with known vulnerabilities, we demonstrate that ExpGen can effectively generate partial exploit input that carries out some address-leakage event and provide a complete automated exploitability evaluation workflow on vulnerable programs running in the ASLR environment.

Keywords: partial exploit; exploit generation; address leakage; runtime exploit relocation

Citation: 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.* **2022**, *12*, 6593. https://doi.org/10.3390/app12136593

Academic Editors: Gregory Epiphaniou, Federico Divina and Carsten R. Maple

Received: 30 May 2022
Accepted: 24 June 2022
Published: 29 June 2022

1. Introduction

Finding bugs in software systems and making on-time repairs is always a critical problem in information security research communities. In the last few decades, with the great enhancement in computation power and constraint solving ability, fuzzing [1,2] and symbolic execution [3,4] have become two mainstream bug-finding techniques in both academy and industry, with derivative techniques including coverage guided fuzzing [5,6], hybrid fuzzing (a composition of fuzzing and concolic execution) [7,8] proposed and effectively applied in bug finding.

With massive crashing inputs generated everyday through different kinds of vulnerability discovery systems, effective automatic vulnerability assessment becomes a must for security analysts as manual analysis is obviously expensive and impossible. Based on taint analysis, tools including !exploitable [9] and HCSIFTER [10] are proposed. Given a specific crashing input, these methods quickly lead the program execution to a crashing point, make some necessary fixes to the corrupted state (e.g., recovering the data corrupted by heap overflow), and then walk further along the state, and report the vulnerability that is exploitable if a taint-based exploitable pattern could be matched. However, as they manifest exploitability without generating exploit input, false positives exist.

Since the Cyber Grand Challenge event [11] DARPA held in 2016, automatic exploit generation(AEG) has become a hot research topic. Solutions including CRAX [12], REX [13], Mayhem [14], are proposed, to discover vulnerabilities and automatically generate exploits when possible, for source code and binary respectively. In general, these methods often

first analyze vulnerabilities in detail along the trace of the crashing input, check out the vulnerability triggering state, then explore further from the vulnerability triggering state to search for program states that are deemed exploitable. Once a exploitable state is derived, these methods collect path-reachability constraints, vulnerability-triggering constraints and exploit-construction constraints separately through symbolic execution, then solve the conjunction of these constraints using SMT (satisfiability modulo theories) [15] solvers, with exploit input finally generated.

However, currently, most AEG solutions can be applied only in exploit generation for vulnerabilities requiring limited exploitation steps in an environment where the Address Space Layout Randomization(abbr. ASLR) [16] vulnerability mitigation mechanism is not deployed. As for vulnerabilities in an ASLR condition, human analysts usually need to leak some important address information from the target process first, then adjust the payload content referencing the leaked address, and finally send the relocated data to the target process to complete the exploitation session. We find that to make this process automatic without human intervention, the following three key challenges should be addressed, which are commonly ignored by current solutions.

Challenge 1: Iterative proof-of-concept generation. Current AEG systems usually first generate some proof-of-concept (abbr. POC) input, then use it to drive the automatic exploit-generation engine to begin its reasoning process. From a systematic perspective, the POC input-generation process and the exploitable state-deduction process are completely separated. This design fits for the 1-step vulnerability exploitation process, where the exploitable state is already implied within the trace of the provided POC input. However, as demonstrated later, exploitability evaluation for vulnerable binary programs running in an ASLR environment often consists of multiple exploitable state-deduction steps, each of which requires a new POC built on capabilities accumulated from the previous exploitation steps. Therefore, we need to introduce bidirectional interaction between the frontend POC-generation process with the backend exploit-generation process, enabling iterative POC generation so as to push forward the exploitation automaton step-by-step till the final state denoting the completion point.

Challenge 2: Balance between address leakage forgery and exploitable state construction. Leaking an address pointing in the range of a payload-relative module is a prerequisite for vulnerability exploitation in an ASLR environment. In the real world, this leakage is usually carefully crafted through the exploitation of an otherwise crashing vulnerability. However, as current AEG solutions usually follow the 1-step exploit-generation philosophy, when they reach the vulnerability-triggering state on which an address-leakage event should be forged, they would simply focus on crafting exploitable states that can directly hijack control flow of the target program, completely ignoring the potential address-leakage artifacts. Some flexible scheduling over the candidate techniques based on the exact exploitation context is critically needed to pull AEG systems out of the dilemma.

Challenge 3: Adaptable payload relocation. As they do not concerning ASLR, current AEG systems often generate exploits working in environments where the address space layout of the target process matches exactly the same with the analysis-time layout exactly. However, this is not the case for the ASLR environment, as the address space layout in runtime is completely different from the one exhibited in analysis time. An extra payload relocation step is needed to fix the gap. However, few current solutions pay attention to this issue.

Our Solution: In this paper, we present ExpGen, a 2-step vulnerability exploitability evaluation system to address the above challenges. Given a binary program with a memory-corruption vulnerability running in an ASLR deployed environment, ExpGen firstly generates a partial exploit that can leak a sensitive address from the target program, then with this partial exploit in mind, performs iterative POC generation seeking to generate some STEP-1 POC inputs that can not only manifest the same address-leakage event, but also direct the target program to reach a new vulnerable state. Based on these STEP-1 POC inputs, ExpGen generates some analysis-time exploit that can leak the desired address information and successfully exploit the vulnerability in the exact analysis-time

environment, and then further performs runtime relocation on the analysis-time exploit. By feeding the relocated payload in the desired interaction order, ExpGen bridges the gap between the analysis-time environment and runtime environment, providing a completely automatic exploitability evaluation workflow with no need for human intervention from security experts.

We have built a prototype of ExpGen based on the state-of-art fuzzer AFL++ [17] and symbolic execution engine S2E [18], and evaluate it on six test programs, 10 CTF&RHG challenges and four real-world applications running under Linux. Experimental results demonstrate that (1) ExpGen can generate partial exploit more efficiently than current test-case generation systems and (2) ExpGen can launch a dynamic exploitation session on all the 20 vulnerable programs in a completely automatic way.

In summary, we make the following contributions in this paper:

- We propose a new iterative partial-exploit sensitive POC-generation mechanism, that can generate 'advanced' POC inputs able to bring in the same leakage event the partial exploit carried out and trigger some new vulnerability, therefore providing support on incremental exploitability evaluation for vulnerabilities requiring multiple exploitation steps.
- We propose an exploitation-context guided analysis-time exploit generation method, which schedules different exploitation techniques according to specific exploitation contexts accumulated in the current exploitable state, providing the ability to either generate a partial exploit that can trigger a forged address-leakage event if no leakage context has been calculated yet, or generate complete a analysis-time exploit input that can carry along all the exploitation steps desired by the real exploitation session.
- We propose a runtime exploit-relocation technique, that can dynamically adapt the analysis-time exploit with precise semantic information to the runtime address space layout, therefore providing a completely automatic exploitability evaluation workflow for vulnerable programs running in an ASLR environment.
- We have implemented a prototype of ExpGen and demonstrated its effectiveness in the exploitability evaluation of the vulnerable CTF&RHG challenges and real-world programs.

2. Motivation Example

In this section, we first illustrate the manual-exploitation workflow for a simple program that contains a stack-overflow vulnerability and runs in an environment where ASLR is deployed, then discuss the limitations of current AEG solutions in handling this case, and finally illustrate the inspirations we draw from the example on the automatic exploitability evaluation of vulnerabilities that require multiple exploitation steps.

2.1. the Vulnerability and Exploitation Process

As for the example program demonstrated in Figure 1a, we compile it with options PIE [19] and NX [20] enabled and run it under Linux-i386 platform with ASLR option turned on. For simplification, we assume the *stack protector* [21] option is turned off during compilation. We can see a function containing some backdoor functionality exists in line 1–3. For brevity we assume in analysis-time the load base of the compiled image being 0x55569000, and the backdoor resides at offset 0x181 in the compiled image.

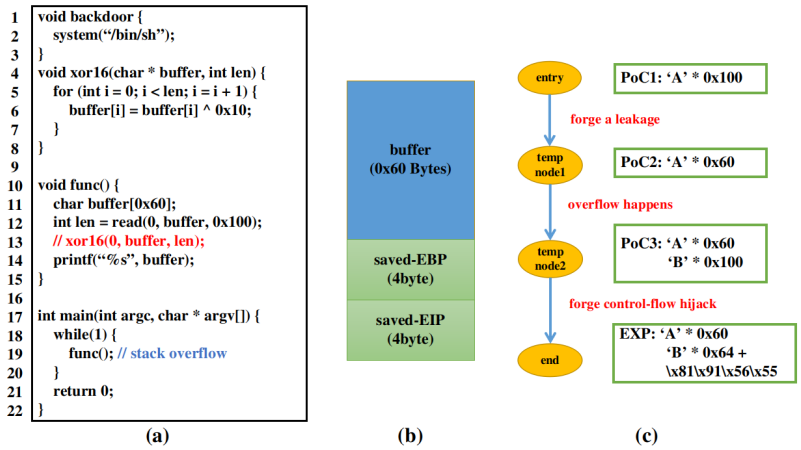

Figure 1. An example of stack overflow exploitation under ASLR environment. (**a**) shows the source code of the example. (**b**) shows the stack frame of func when being called by main. (**c**) shows the exploitation steps carried out with corresponding POC or exploit data shown on the right.

We can also see, in line 12, there exists a call to posix-API function read, which would fetch at most 0x100 bytes from the standard input channel to the local array pointed by the buffer, whose size is only 0x60, resulting in a potential stack-overflow vulnerability. From the stack layout demonstrated in Figure 1b, we can see that, when the program receives more than 0x60 bytes, the saved EBP or, what is more important, the saved EIP, would be overwritten, therefore triggering a segmentation fault when the called function func returns. We temporarily comment out line 13, which will be discussed later in Section 2.2.2.

The manual exploitation process of the example program is demonstrated in Figure 1c, consisting of the following phases.

1. Address-leakage exploit generation. We craft an input S_0 that would take up precisely 0x60 none-zero bytes. When fed to the target program, this input would not overflow the local variable buffer in func's stack frame, but can cause the call on printf at line 14 to leak extra data on the stack including the saved EBP and the saved EIP(as the EBP value usually consists of four none-zero bytes under 32 bit systems at runtime). Therefore we can obtain a leaked address of the main executable which is the return address of the callage to func in main. S_0 would be the first data sent to the target program during the attack session.

2. Control-flow hijack exploit generation. We generate an input S_1 that can trigger function backdoor, which would not be invoked under normal conditions. S_1 is composed of 0x64 none-zero bytes (note, here we pad saved EBP with 4 bytes) trailed by 0x55569181, which is backdoor's address at analysis time. Thus, when the control flow returns from func to main, we can hijack the control flow onto backdoor and obtain a shell from it.

3. Exploit synthesis. As we have got two exploits, separately S_0 capable of leaking a runtime address in the main executable and S_1 capable of control-flow redirection onto backdoor in the main executable at analysis time, we now synthesize the two crafted inputs into a complete one in the following dynamic exploitation session.

When the target program is launched, we feed it with S_0 through the standard input channel first, and fetch the leaked runtime address $A_{runtime}$ by monitoring the byte stream from the standard output channel. With the corresponding analysis-time address $A_{analysis}$ in mind, we replace the analysis-time specific value 0x55569181 in S_1 with $A_{runtime}$-$A_{analysis}$ + 0x55569181, obtaining a new input block S_1' where the target program-address-space related fields are all 'relocated' to the runtime environment. Lastly, we send S_1' to the target process with the desired shell successfully obtained afterwards.

2.2. Inspirations for AEG

This section performs a thorough introspection of the general workflow of current AEG solutions, analyzes the reasons why they cannot handle the above example, and

then proposes several principles we believe critical to enabling AEG systems to evaluate exploitability for vulnerabilities requiring multiple exploitation steps.

2.2.1. General Workflow of Current AEG Solutions

As mentioned earlier, current AEG solutions generally follow the workflow demonstrated in Figure 2. Given the target program and a seed corpus, current AEG systems first employ state-space exploration techniques including fuzzing, symbolic execution, etc, generating massive test cases to explore the path space of the target program as much as possible. If some inputs are found able to trigger a crash, or violate some security properties enforced by sanitizers such as AddressSanitizer [22], etc.), these input would be deemed to be a POC input manifesting a vulnerability and be provided directly to the backend dynamic trace-analysis engine.The dynamic trace-analysis engine traces the execution of the target program under the specific POC input until the desired vulnerability is triggered, then makes necessary modifications to on that program state to drive the execution further to reach an exploitable state, e.g., EIP overwritten. If an exploitable state is discovered, the exploitation technique scheduler enumerates known exploitation techniques including ret2libc [23], ret2text [24], etc., to build a exploitable memory layout, on which the path-reachable constraint, the vulnerability triggering constraint and payload-arrangement constraint are collected. By calculating a concrete input that satisfies the conjunction of the three constraints, an exploit input for the vulnerability is finally generated.

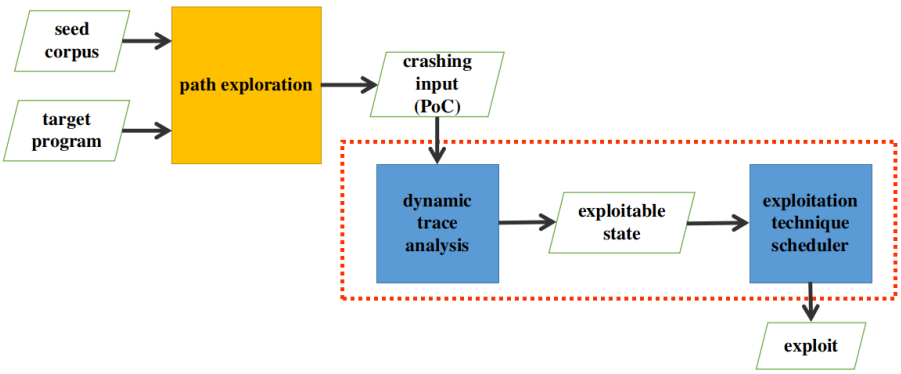

Figure 2. The general workflow of current AEG systems.

2.2.2. Principles for Automatic Multi-Step Generation

Though current AEG systems can already generate exploits for memory-corruption vulnerabilities, including stack overflow, heap overflow, and format-string vulnerability, effectively under environments where exploitation-mitigation mechanisms, including NX and stack canary, are deployed, for the demo example shown in Figure 1, they often fail in exploit generation. However, we can derive some key insights from the example, which we believe could be used as principles for advanced AEG system design and implementation. These principles are listed below.

principle 1. Formulation of 2-step exploitation automaton. As we can see in Section 2.1, the exploitation process for the example program consists of two steps, separately one for address leakage, the other for control-flow hijack. Though the target program has an evident stack-overflow vulnerability, we should not trigger it immediately when a crashing event happens, but forge an address-leakage event instead. Only when this prerequisite is met can we push forward the exploitation-evaluation session onto the next stage, where we generate an exploit able to pwn at analysis time. However, as mentioned earlier, current AEG systems generally follow a one-step exploitation philosophy. When reaching a crashing point, they would immediately seek an exploitable state, irrespective of the intermediate leakage step that must be completed before. Therefore, we can conclude that the formulation of a 2-step exploitation automaton is a must to solve this issue.

principle 2. An iterative POC-generation mechanism with a partial-exploitation context in mind. As the exploitation process for complex vulnerabilities usually consists of multiple steps, deriving the final exploitable state directly through a single POC input discovered immediately by some blind fuzzer becomes no longer feasible, as it is hard for such a POC input to trigger all the desired exploitation effects. Therefore, a new iterative POC-generation mechanism should be devised to discover new exploitation steps and accumulate them onto the POC invariant held during each iteration, driving the multi-step exploitation automaton to continuously move forward.

principle 3. Semantic-information-based fix on the gap between analysis-time exploitation and runtime exploitation. As for exploitability evaluation on vulnerabilities in environments where ASLR is enabled, during manual exploitation, analysts should perform runtime relocation on the module-related payload after leaking some sensitive address, in order to ensure that the control flow of the target program can be hijacked gracefully without triggering a segmentation fault. To our knowledge, among current AEG solutions only Marten [25] has paid attention to the automatic anatomy. Through the leaked address, Marten calculates the load base of the module the leaked address resides in, and then adds it to the corresponding payload part. For brevity, we call this literal relocation as it works directly on the literal form of input words. However, for programs involving complex computations, literal relocation often fails due to a lack of semantic information.

For demonstration, let us review on the example program. This time, we uncomment line 13 in Figure 1a, thus, the compiled program performs an extra exclusive or operation on the input content. We now focus on the payload part Exp_{EIP} that would overwrite EIP in runtime. To overwrite EIP with 0x55569181, which is *backdoor*'s analysis time address, Exp_{EIP} would be evaluated as '\x91\x81\x46\x45' in little endian by the analysis engine. Now, assume that, at the exploitation time we derive the load base of the main executable to be 0x80480000 from the leaked address, using literal relocation, we should adjust Exp_{EIP} with '\x91\x81\x8e\xc5'. However, when we feed the relocated payload to the target program wishing to obtain a shell, the program would simply crash as the overwritten EIP at runtime was mistakenly pointing onto '\x81\x91\xe9\xd5'(0xc58e8191 XOR 0x10101010 = 0xd59e9181), which is an unmapped address.

This failure shows that blindly applying literal relocation to a gap between analysis-time and runtime is not sufficient in AEG for complex programs; only considering fine-grained semantic knowledge in concern can we provide a more complete solution.

3. Design of ExpGen

Based on the derived principles listed in Section 2.2.2, we propose ExpGen, a 2-step vulnerability-exploitability evaluation solution for binary programs in environments where the ASLR vulnerability mitigation technique is adopted. As demonstrated in Figure 3, ExpGen is composed of three major components, namely, partial-exploit sensitive POC generation, analysis-time exploit generation, and runtime exploit relocation.

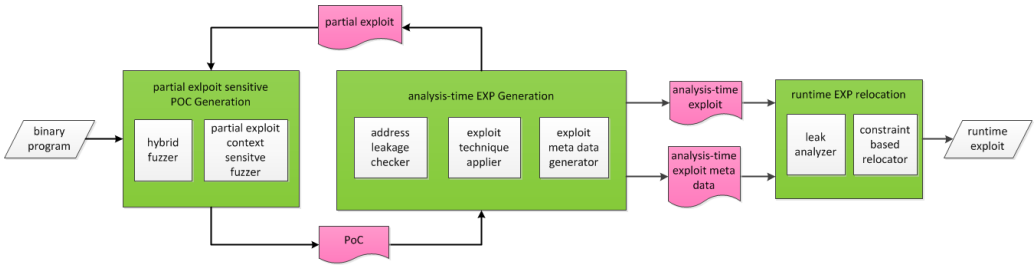

Figure 3. Overview of ExpGen.

Given a binary program, ExpGen first employs the POC-generation engine generating POC inputs that can lead the execution of the target program trigger a vulnerability. These POC inputs are then fed to the analysis-time-exploit generation engine, which, with the exploitation automaton in mind, tries to generate a partial exploit that can leak some

address information from the target program. This partial exploit is then proposed back to the POC-generation engine, driving the latter to generate a STEP-1 POC input not that only holds the desired partial-exploit context, but also triggers a new vulnerability behavior. This STEP-1 POC input is then used to drive the analysis-time EXP-generation engine to move a new exploitation step forward, on which we finally obtain the analysis-time exploit which embodies the complete exploitation steps and the corresponding meta data holding a description of this analysis-time exploit. These two outcomes are then fed to the backend runtime exploit generation engine, which analyzes the leakage scheme and performs dynamic payload relocation on the analysis-time exploit during the runtime exploitation session. If the target system's control flow is successfully hijacked, the vulnerability is deemed exploitable in an ASLR environment by ExpGen.

4. Partial-Exploit-Sensitive POC Generation

As depicted in Figure 3, ExpGen builds its partial-exploit-sensitive POC -generation engine upon a hybrid fuzzer, then constructs partial-exploit-sensitive fuzzing atop it. This section illustrates the details.

4.1. Hybrid Fuzzer

As demonstrated in Figure 4, The hybrid fuzzer is composed of two components, a feedback-directed fuzzing engine and a generation-based symbolic execution engine.

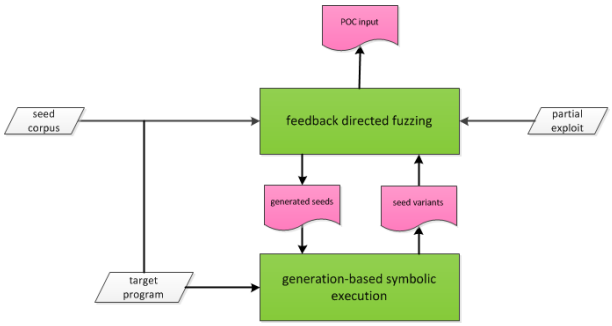

Figure 4. Overview of hybrid fuzzer.

The feedback-directed fuzzing engine maintains a coverage bitmap recording the coverage information on each edge in the target program, collects the execution statistics of a given seed through dynamic execution monitoring, and then checks if this seed can hit new edges not covered before. If so, the coverage bitmap is updated, with the seed input considered as a promising one and added to the promising queue holding test cases on which fuzzing engine would perform further mutations later on to explore new paths in the target program.

The generation-based symbolic-execution engine takes the target program and a single seed as input, calculates a corresponding path constraint by concolicly tracking the execution trace, and then generates new constraints by flipping each condition in the path constraint. Through querying these constraints, some variants would be derived.

Our hybrid fuzzer combines the above two components in a bidirectional-enhancing way. The test cases deemed promising by the feedback-directed fuzzing engine are provided as candidate seed inputs to the generation-based symbolic execution, enabling the latter to explore different execution paths while avoiding the path-explosion problem commonly faced by symbolic execution techniques. The generation-based symbolic execution, on the other hand, generates variants of the seed input that can exercise different execution paths. These variants are synchronized by the feedback directed fuzzing engine, which would check out those variants that can be deemed as promising and start a new round of test-case generation.

4.2. Partial Exploit Sensitive Fuzzing

Once some partial exploit carrying out an address leakage-event is generated by the back end analysis-time exploit generation engine, as depicted in Section 5, the above hybrid fuzzer has to generate test cases that not only exhibit the same address leakage effect exposed by the partial exploit, but also trigger a new vulnerability afterwards. To implement this kind of partial-exploit sensitive fuzzing, we make the following adaptions to both the feedback-directed fuzzing engine and the generation-based symbolic execution engine.

On the feedback-directed fuzzing engine side, we make address leakage an additional feedback source. More specifically, we collect the address space layout information during the execution monitoring phase, along with implementingdynamic hooks on API functions including write, recv, etc, that would send data to the output channel. An example hook on a POSIX write function in Linux i386 platform is demonstrated in Figure 5. After checking the output file descriptor pointing to our desired channel (line 7–9), this hook extracts the buf parameter and nbytes parameter denoting the memory region to write (line 11–13), and checks if this region contains some address pointing within the memory region of the main executable or the libc.so module, which is the most common runtime library in the GNU Linux platform (line 15–17). If some address information is located in the output buf, we set the corresponding leakage context (line 18–24) for this test case, marking it as definitely promising for further generating a test case upon it.

```
1// write(int fd, const void * buf, size_t nbytes);
2static int write_call_hook(CPUX86State * state) {
3    target_ulong fd = 0;
4    char * str_ptr = 0;
5    target_ulong size = 0;
6
7    fd = *((target_ulong *)g2h(state->regs[R_ESP] + sizeof(target_ulong)));
8    if (!is_output_channel(fd)) {
9        return 0;
10   }
11   str_ptr = *((target_ulong *)(g2h(state->regs[R_ESP] + sizeof(target_ulong) * 2)));
12   str_ptr = (char *)(g2h(str_ptr));
13   size = *((target_ulong *)g2h(state->regs[R_ESP] + sizeof(target_ulong) * 3));
14
15   char * modName = NULL;
16   target_ulong modOffset = 0;
17   int leak_idx = check_modAddr_in_string(str_ptr, size, &modOffset, &modName);
18   if (leak_idx != -1) {
19       if (1 == is_main_executable(modName)) {
20           set_vulcontext(ADDR_LEAK_MAINIMAGE);
21       } else if (1 == is_libc(modName)) {
22           set_vulcontext(ADDR_LEAK_GLIBC);
23       }
24   }
25   return 0;
26}
```

Figure 5. Hook on write.

On the generation based symbolic execution side, given a test case *tcase* generated by feedback-directed fuzzing based on a partial exploit input $exp_{partial}$, assumeing that *tcase* would lead the single-path symbolic execution to obtain a path constraint c_0 AND ... AND c_i AND c_{i+1} AND ... AND c_n, where conditions ranging from c_0 to c_i are derived by the prefixing $exp_{partial}$ part and conditions ranging from c_{i+1} to c_n are derived by the remaining part in *tcase*, the generation-based symbolic execution engine would only flip the latter conditions, while keep the former unchanged in test-case generation.

5. Analysis-Time EXP Generation

Given a specific POC input, the analysis-time EXP generation component performs concolic analysis along the corresponding trace of the target program. During this process, ExpGen actively records the leakage events that happened alongside the trace, and upon discovering an exploitable state, either tries to forge a leakage event if no leakage context has yet been recorded so far, or directly applies known exploitation techniques based on the maintained leakage context, generating an analysis-time exploit that can carry out all the necessary exploitation steps deriving complete exploitation of the vulnerable program in the exact analysis-time environment.

5.1. Address-Leakage Checker

During the runtime analysis of a specific execution, the address leakage checker component checks if the execution trace constitutes address-leakage events. If so, the corresponding leakage information would be recorded in the exploitation context, providing references for exploitation technique applications when some exploitable state is reached. In practice, we monitor three types of address-leakage events, namely, explicit address leakage, implicit address leakage and forged address leakage.

Explicit address leakage denotes the situation where the execution directly sends some address information to its output stream. This time, we record the leaked address $addr_{leak}$ in the corresponding context, and keep the execution continue until an exploitable vulnerability is encountered.

Implicit address leakage refers to occasions where the above explicit leakage does not directly happen, but either the pointer of the output buffer or the origin of the output content can be controlled. Take the program snippet shown in Figure 6 for example. Given a POC input containing 1024 '\0', the execution of the target program would pass through the callage on printf at line 10. However, as we find the index variable that determines the pointer on the output content can be influenced by the POC input during symbolic execution, in such a case, we can bind address pointing to some known regions that contain sensitive address information (e.g., the global offset table [26]) on the symbolic pointer $data + index \times 4$ to dynamically turn this implicit leakage into an explicit one.

```
1 #include <stdio.h>
2
3 char data[1024];
4
5 int main(int argc, char * argv[])
6 {
7     char buffer[10];
8     read(0, buffer, 10);
9     int index = *((int *)(buffer));
10    printf("content: %d", data[index]);
11    read(0, buffer, 1024);
12    return 0;
13 }
```

Figure 6. A Demostration Example on Implicit Leakage.

Forged address leakage refers to states that would normallybe terminated due to exception; however, under a certain transformation that breaches the consistency of the execution state, they can be transformed into a partial exploitable state. Take Figure 1b for example. Assume, in analysis time, we drive the target program with a POC input containing 0x100 'A's. Although this input would invoke the stack-overflow vulnerability at line 12, triggering a sensitive address write operation on EIP when returning from function $func$ at line 15, we should not directly generate the final exploit input on the exploitable site as we have not leaked any address layout information to bypass ASLR yet. However, as the call on 'printf' at line 14 would print a string starting at an address pointing to the stack frame of the function 'func' and the string content is controllable by the external input, we can truncate the length of the string starting at buffer to 0x60 at that state, and therefore turn the original crashing stack overflow into a forged information leakage that would happen without triggering a segmentation fault. As the truncation would breach the data consistency of input length, we generate a partial exploit input that would trigger the forged information leak in the truncated context, and hope the POC-generation engine described in Section 4 generates a new POC input that would trigger some new vulnerability while absolutely triggeringthe forged leak event embodied by the partial exploit.

5.2. Exploitation-Technique Applier

During dynamic analysis, our exploitation technique applier monitors four types of exploitable states, separately onStackEipHijack, onFuncPtrOverwrite, onFuncContentOverwrite and onStackEBPHijack. Table 1 demonstrates the related context that would be recorded in these states. When ExpGen discovers some exploitable state, the corresponding context information is recorded. The exploit-technique applier then schedules the available known exploitation techniques bound to this exploitable state, trying to generate an analysis-time exploit input that can carry out all the necessary steps in complete exploitation process. The bounds between the exploitable states and exploitation techniques are shown in Figure 7.

Table 1. Specifications on Exploitable States.

Exploitable State	Contexts to Record
onStackEipHijack	the overwritten return pointer on stack, ESP and sequence of symbolic bytes from ESP
onFuncPtrOverwrite	the content of the overwritten function pointer
onFuncContentOverwrite	the function pointer pointing the the overwritten content and the size of the overwritten content
onStackEBPHijack	the overwritten EBP on stack

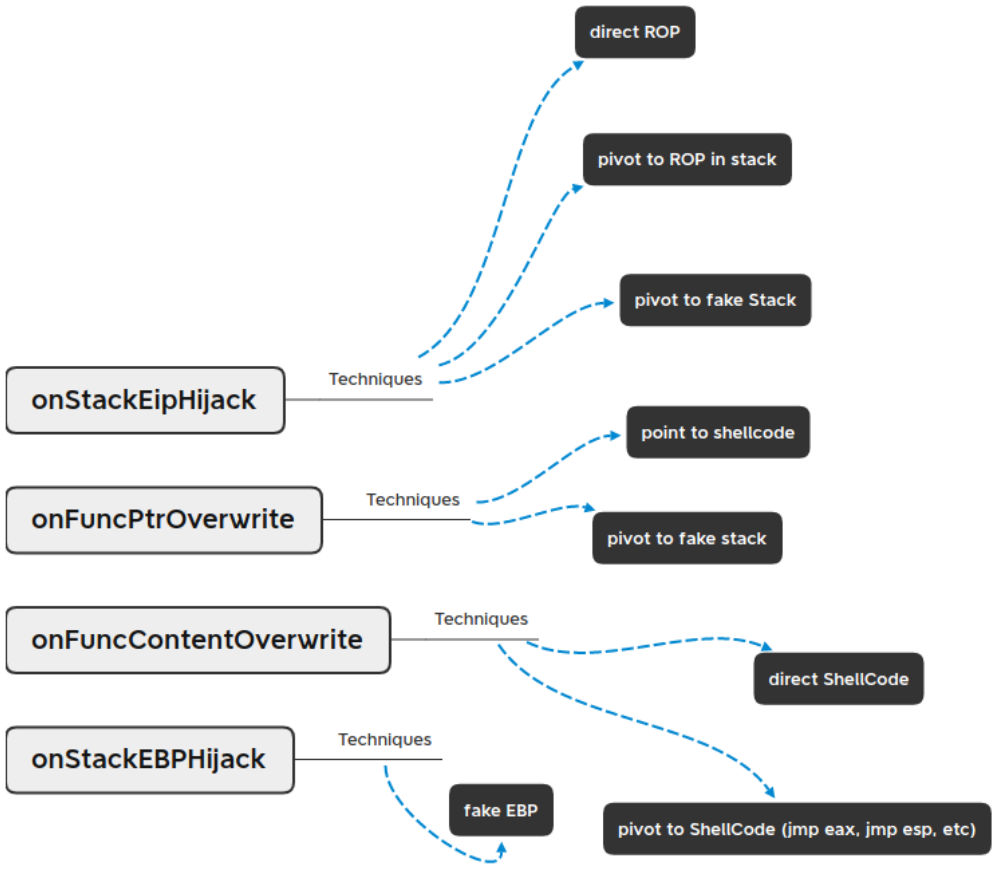

Figure 7. Bounds between Exploitable States and Exploitation Techniques.

5.3. Exploit-Meta-Data Generator

As mentioned in Section 2.2.2, the gap between analysis-time exploit generation and runtime attack session should be fixed in order to make the exploit input adaptable to runtime exploitation. ExpGen separates this process into two stages, separately, a meta data generation phase leveraging description information of the analysis-time exploit, and

a runtime-exploit relocation phase performing the semantic-aware exploitation transplantation. We introduce the former phase in this section, leaving the latter to Section 6.

Speaking more precisely, the exploit meta data should contain the necessary semantic information necessary on the address space-layout-related parts in the generated exploit. Therefore, during analysis time, we collect the address-space layout information containing each module's load base and size, the range of the stack region and the heap region. If an address leakage happens, the leaked address is also collected. If some analysis-time address is utilized by the exploitation-technique applier when scheduling some payload arrangement techniques, the exact address is also recorded together with the final query string of the complete SMT exploit constraint for later off-line relocation.

6. Runtime-Exp Relocation

When an exploit input is generated together with the corresponding meta data by the analysis-time EXP generation engine, ExpGen employs runtime-EXP relocation to make it adaptable to the real-world environment under evaluation. As demonstrated in Figure 3, the runtime-EXP relocation consists of two parts, separately a leak analyzer and a constraint-based relocator. This section illustrates how these two components work together to fix the gap between analysis-time exploit generation and runtime vulnerability exploitation.

6.1. Leak Analyzer

The main duty of the leak analyzer is to calculate the interaction sequence which the runtime exploitation process should follow, and information about the leaked address including the offset in the output stream, the output format, etc. We discuss how the leak analyzer completes the above calculation through two dynamic analysis methods, separately, an IO-sequence analysis and a leak-location analysis.

6.1.1. Io Sequence Analysis

Given a specific analysis-time exploit input *seed*0 and a target program P, an IO sequence defines the input/output actions that should happen during the runtime-exploitation process on P with *seed*0 provided as the source of the input stream. For the simple program demonstrated in Figure 1a, assume an analysis-time exploit $Exp_{analysis}$ containing two parts, separately one with 0x60 'A's and another with 0x64 'B's suffixed by 0x81915655, the IO sequence $IOSeq$ for $Exp_{analysis}$ is demonstrated below:

$$\begin{aligned} IOSeq = \{&IOEntry_1 = \{\text{type = READ, len = 0x60}\},\\ &IOEntry_2 = \{\text{type = WRITE, len = 0x68}\},\\ &IOEntry_3 = \{\text{type = READ, len = 0x68}\},\\ &IOEntry_4 = \{\text{type = WRITE, len = 0x68}\}\\ \}& \end{aligned}$$

Based on an open-sourced whole-system dynamic analysis platform, we filter out the specific execution context for the target process through process monitoring, record the file descriptors allocated for the IO stream, then dynamically hook the read and write system calls that happened in the execution context. Once a read/write action happens on the desired file descriptor, the corresponding information is recorded in the above form. When the target process terminates, the complete sequence is recorded for further exploitation.

6.1.2. Leak-Location Analysis

Given the output stream of the target program, ExpGen employs leak-location analysis, a dynamic analysis aiming to determine the range in the stream corresponding to a leaked address, together with the format of the leaked content. With this information in mind, the back-end-constraint-based relocator can dynamically extract the desired leaked addresses at runtime for payload relocation.

The leak-location analysis is implemented through a customized pwntool [27] script. When the main executable is loaded into the process, our leak-location analysis script collects the complete address-space layout information, and then interacts with the target process in the exact order specified by the IO sequence derived in Section 6.1.1. During the interaction, the leak-location analysis retrospects the output stream byte by byte in three patterns, separately, the literal form, hexadecimal form and string form, trying to extract some pointer-typed content pointing exactly to a module already loaded in the address space. If successful, the desired information is gathered together for later use.

6.2. Dynamic Exploitation with Constraint-based Relocation

At the final stage, ExpGen synthesizes all the auxiliary information mentioned above, transferring the analysis-time exploit into a well-suited one for the running process during the exploitation session. Algorithm 1 shows the complete process.

Algorithm 1 Runtime Exploitation Session with Constraint Based Payload Relocation

Input:
The full duplex channel to connect to the target process, $proc$.
The analysis-time exploit input, $Exp_{analysis}$.
The meta data on $Exp_{analysis}$, $ExpMd$.
The IO sequence, $IOSeq = \{IOEntry_1, IOEntry_2, \ldots, IOEntry_n\}$, where $IOEntry_i (1 \leq i \leq n)$ is the record describing the ith IO action.
The leaked address descriptions, $LADescs = \{LADesc_1, LADesc_2, \ldots, LADesc_n\}$, where $LADesc_i$ $(1 \leq i \leq n)$ characterizes the ith leaked address in the output stream.
Output:

```
1:  smtSolver = SMTSolver()
2:  contents = Exp_analysis
3:  sendIndex = 0
4:  recvIndex = 0
5:  LAIndex = 0
6:  for each IOEntry ∈ IOSeq do
7:    if IOEntry.type == READ then
8:      sendIndex_end = sendIndex + IOEntry.len
9:      proc.send(contents[sendIndex : sendIndex_end])
10:     sendIndex = sendIndex_end
11:   else if IOEntry.type == WRITE then
12:     recvContent = proc.receive(IOEntry.len)
13:     if recvIndex <= LADescs_LAIndex.offset AND IOEntry.len + recvIndex >=
        LADesc_LAIndex.offset + LADesc_LAIndex.len then
14:       leakAddr = extractLeakAddr(recvContent,
15:                                  LADescs_LAIndex.offset − recvIndex,
16:                                  LADescs_LAIndex.len, LADescs_LAIndex.fmt)
17:       modBase = leakAddr − LADescs_LAIndex.modOffset
18:       analysisAddrs = ExpMd.addr_slots[LADescs_LAIndex.modName]
19:       actualAddrs = relocAddrs(analysisAddrs, modBase)
20:       smtConstraint = rewriteConstraint(ExpMd.smtConstraint,
21:                                         analysisAddrs, actualAddrs)
22:       content = smtSolver.query(smtConstraint)
23:       LAIndex = LAIndex + 1
24:     end if
25:     recvIndex = recvIndex + IOEntry.len
26:   end if
27: end for
```

Given the full duplex channel proc connected to the target process, the complete exploit $Exp_{analysis}$ generated in analysis-time, the corresponding meta data $ExpMd$, the IO sequence $IOSeq$, and the leaked address descriptions $LADescs$, the algorithm arranges

the input/output actions in the exact order specified by *IOSeq*. When a READ action should be performed, the corresponding region in content is located and then sent to the target process through the *send* interface of proc. When a WRITE action is on demand, that is, the target process sends back some data to us, we fetch the data through the receive interface, and then check if an address leakage exists in by referencing description $LADescs_{LAIndex}$ of the current anticipated leak address at line 13. If it is confirmed to exist, the actually leaked address is calculated at line 14–16, on which we can obtain the corresponding load base of the module M in which the leaked address resides (line 17). This module base is then utilized to calculate the corresponding runtime addresses actualAddrs for all the address values analysisAddrs used in analysis-time exploit generation that resides in the same module at analysis time (line 18–19). We then derive an updated exploit constraint by substituting analysisAddrs to actualAddrs in ExpMd.smtConstraint (line 20–21). Through querying on this constraint, we can generate a new exploit (line 22) correctly fixing the gap for module M between analysis-time and runtime. For ongoing IO interactions, this updated exploit (line 21) would be used as source of input stream; therefore, the updated exploit can be sent to the target process, executed normally in the runtime address space-layout.

7. Evaluation

We implemented a prototype of ExpGen based on the fuzzer engine AFL++ [17], whole-system binary symbolic execution engine S2E [18], SMT constraint solver Z3 [28], and the exploit development library pwntools [27]. It consists of 2001 lines of C++ code on the S2E side and 7804 lines of C code on the AFL++ side to construct the partial-exploit-sensitive POC-generation component, 14,512 lines of C++ code on the S2E side generating the analysis-time exploit, and 1022 lines of Python code synthesizing utilities provided by pwntools and Z3 implementing dynamic exploitation on the vulnerability in the remote process with constraint-based relocation performed on time.

In this section, we present the evaluation results of this system. These experiments weer all carried out in a 64-bit Ubuntu 20.04.3 LTS system on a workstation with 64 G RAM and Intel(R) i9-9880H CPU @ 2.30 GHz.

7.1. Benchmarks

The benchmarks we constructed to evaluate our system are demonstrated in Table 2. These benchmarks are composed of six test programs, four programs from RHG 2021ljb challenge [29] held in China in November 2021, two programs from RHG 2021 challenge [30] held in China in March 2021, four challenges from CTF events and four real-world applications with exposed vulnerabilities. These benchmarks are all 32 bit ELF user-mode applications running under i386 architecture machines. These programs were collected based on the following criteria.

- The benchmark programs should cover different types of vulnerabilities. As we can see in Table 2, there are five types of memory-corruption vulnerabilities, separately, integer overflow, format string, use after free, heap overflow and stack overflow, covered in this collection. These are all memory-corruption vulnerabilities commonly seen in binary programs.
- The benchmark programs should run in environments where ASLR is deployed, and leaking an address is a must in the vulnerability-exploitation process. As for the latter, there are two types of address leakage, separately, explicit leakage and forge-needed leakage. In the former cases, the target program would intentionally leak an address to the output channel, while, in the latter cases, when there exists no such direct information leakage, the analysis engine has to forge an address leakage event utilizing the vulnerabilities in the target program so as to carry out the complete exploitability evaluation process.
- The real-world applications are mainly from user-mode applications with public vulnerabilities. These programs take input either from the command line arguments,

the environment variables or the standard input channel, sending output directly through the standard output channel.

Table 2. Evaluation Benchmark.

Dataset	Program	Vulnerability Type	Leakage Type
test programs	vul1	integer overflow	forge-needed
	vul2	integer overflow	explicit
	fmt1	format string vulnerability	explicit
	fmt2	format string vulnerability	explicit
	heap1	use after free	explicit
	heap2	heap overflow	explicit
RHG 2021ljb	question5	stack overflow	explicit
	question6	stack overflow	explicit
	question8	format string vulnerability	forge-needed
	question13	stack overflow	forge-needed
RHG 2021	prob1	stack overflow	forge-needed
	prob8	format string vulnerability	forge-needed
plaidctf-2013	ropasaurusrex	stack overflow	forge-needed
bamboofox	ret2libc3	stack overflow	forge-needed
pwnable.tw	start	stack overflow	forge-needed
	unexploitable	stack overflow	forge-needed
real applications	ncompress 4.2.4 (CVE-2001-1413)	stack overflow	forge-needed
	iwconfig v26 (BID-8901)	stack overflow	forge-needed
	DNSTracer 1.9 (CVE-2017-9430)	stack overflow	forge-needed
	rsync 2.5.7 (CVE-2004-2093)	stack overflow	forge-needed

We conducted experiments trying to answer the following two research questions, with some results published at https://github.com/helios-ops/ExpGen.git.

- RQ1: Can ExpGen generates partial exploits that can leak a sensitive address locating in the address space layout of the target programs more efficiently and effectively than current test-case generation methods?
- RQ2: Can ExpGen generates an exploitation session able to exploit the target programs in environments where ASLR is deployed, when crafting an address leakage is a must in exploitation?

7.2. Effectiveness in Partial Exploit Input Generation

To answer RQ1, we compared ExpGen to open-sourced test-case generation engines. These engines include AFL, AFL++, which are systems based on fuzzing, S2E, a state-of-art symbolic execution engine, and QSYM, a well-known hybrid fuzzer. To determine whether ExpGen could perform better than current systems in generating a partial exploit that can successfully leak some runtime address information when fed to the target program, we evaluated these systems under benchmarks whose leakage type is 'forge-needed' in Table 2.

We provide the same initial seed corpus to each engine in the conducted experiments. We also allocate for each engine the same computation resources including the processor cores, the memory limit, the time limit, etc. As ExpGen combines one AFL instance and two S2E instances to generate partial exploits, we customized the other engines mentioned above with the same computation capability including three logical processors, 64 GB memory with a time quota lasting for 2 h. The specific configuration for each engine is demonstrated in Table 3. We also provide an introspection process to check whether the generated test cases can execute the same path executed by the desired partial exploit, or,

what is more, leak some address information when fed to the target program. For the latter case, the time cost in partial exploit generation was also calculated at runtime.

Table 3. Engine Configuration.

Engine	Configuration Setup
ExpGen	1 AFL++ instance, 2 S2E instances
AFL	1 AFL master instance, 2 AFL slave instances
AFL++	1 AFL++ master instance, 2 AFL++ slave instances
S2E	3 S2E instances working in parallel
QSYM	1 PIN-based concolic executor, 1 AFL master instance, 1 AFL slave instance

Figure 8 demonstrates the results. We can see that ExpGen is able to generate partial exploits for the all list programs much more efficiently than other test-case generation engines. We believe the main reason for this phenomenon lies in the exploitation awareness uniquely exhibited by ExpGen. As for fuzzers such as AFL and AFL++, they generally generate cluster-program inputs by the branch-coverage statistics. They often represent a branch coverage by the first test case they encountered during generation time, discarding other test cases that exhibit the same branch coverage. This philosophy does not work for partial-exploit generation. As we can see from Figure 1, the path taken by the desired partial exploit can be covered easily by test cases generated through some simple data mutation. Thereby though carrying out an evident address-leakage feature, the exact partial exploit would simply not be viewed as promising, missing the opportunity to participate in the further round of test case generation. For S2E and QSym, their core symbolic-execution parts classify each input by the path constraint it takes, also neglecting the exploitation context information during test-case generation. ExpGen, however, because of its address leakage sensitivity, can definitely identify out the interesting exploitation step carried out by the partial exploit, marking it as a promising seed for further test-case generation, therefore obtaining better results on this criterion.

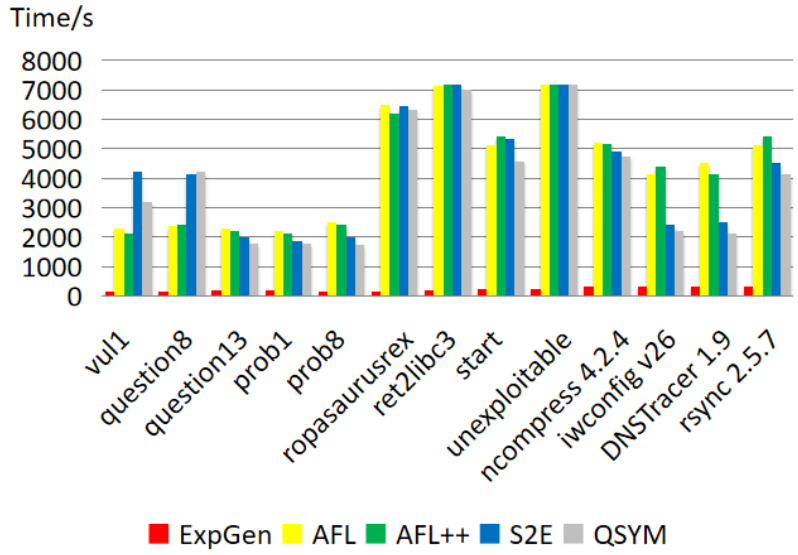

Figure 8. Time Cost in Partial Exploit Generation of each Engines.

7.3. Effectiveness in Exploit Generation

To answer RQ2, we compared ExpGen with REX [13], an open-sourced automatic exploit-generation engine developed by the Shellphish team. We choose from benchmarks

listed in Table 2 that exhibits both 'explicit' and 'forge-needed' in leakage type, feed them with partial exploits that can already cause an address-leakage event as the initial seeds, test if they can generate exploit input at analysis-time and further try to perform complete exploitation on the target programs.

Table 4 demonstrates the results. We can see that although REX can generate analysis-time exploit successfully as ExpGen given the same partial exploit already embodying the desired address leakage effect, it fails in actual exploitation on all the programs listed in the benchmark, while ExpGen achieves complete success. We believe the main reason for this huge difference lies in the novel runtime exploit relocation technique we outline in this paper. Working as ExpGen's back end, the runtime-exploit relocation engine dynamically fixes the gap between the analysis-time address space layout and runtime address space layout, therefore providing a unique advantage for ExpGen in complete automation of the exploitability evaluation process.

Table 4. Experimental results of exploit generation.

Program	Engine	Exploit Generation	Leakage Invoked	PWN Ability
vul1	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail
vul2	ExpGen	ok	explicit	success
	REX	ok	explicit	fail
fmt1	ExpGen	ok	explicit	success
	REX	ok	explicit	fail
fmt2	ExpGen	ok	explicit	success
	REX	ok	explicit	fail
heap1	ExpGen	ok	explicit	success
	REX	ok	explicit	fail
heap2	ExpGen	ok	explicit	success
	REX	ok	explicit	fail
question5	ExpGen	ok	explicit	success
	REX	ok	explicit	fail
question6	ExpGen	ok	explicit	success
	REX	ok	explicit	fail
question8	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail
question13	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail
prob1	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail
prob8	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail
ropasaurusrex	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail
ret2libc3	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail
start	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail
unexploitable	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail
ncompress 4.2.4	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail
iwconfig v26	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail
DNSTracer 1.9	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail
rsync 2.5.7	ExpGen	ok	forge-needed	success
	REX	ok	forge-needed	fail

8. Discussion

This paper proposed ExpGen, a vulnerability exploitability evaluation solution for binary programs running in environments where the ASLR vulnerability-mitigation technique is adopted. Different from current AEG solutions, which generally follow an one-step exploit generation philosophy immediately crafting exploitable states that can directly hijack control flow when reaching a vulnerability-triggering state, ignoring the intermediate stages necessary to leak some address information from the target program's address space layout, ExpGen takes an iterative automation methodology employing three novel techniques, separately, partial-exploit-sensitive POC generation, exploitation-context-sensitive analysis-time exploit generation and dynamic runtime exploit relocation.

The partial-exploit-sensitive POC-generation mechanism constructs a bidirectional enhancing hybrid fuzzer containing feedback-directed fuzzing and generation based symbolic execution. On the feedback-directed fuzzing engine side, we make address leakage an additional feedback source. On the generation-based symbolic-execution engine side, we only flip the path conditions not embodied by the partial exploit, while keep the remainder unchanged.

The exploitation-context-sensitive analysis-time exploit generation actively records the leakage events happening during dynamic concolic execution, mimics known exploitation techniques and schedules them according to the specific exploitable state met during analysis, generating analysis-time exploit input that can carry out all the desired exploitation steps in the analysis-time environment. This analysis-time exploit is then fed to the runtime exploit-relocation engine, which by performing semantic aware payload relocation, makes analysis-time exploit dynamically adaptable to the runtime exploitation session.

As demonstrated in Section 7, ExpGen obtains better results in partial-exploit generation than current test-case generation systems and is proven effective in automatically evaluating exploitability on vulnerable programs running in the ASLR environment. However, there are still improvements that can be made on ExpGen. We list them below for future works.

- Semantic-aware payload generation. AEG systems generally takes shellcode or ROP sequence as payload to execute. When an exploitable state is discovered, the imitated exploitation techniques all try to direct the hijackable control flow onto these payload. However, as for current AEG systems, these payloads are often provided beforehand. This would often fail when the payload to be constructed must follow a specific constraint imposed by the program semantics. The automatic crafting of the desired payload in the exploitable state with program semantics information in mind is definitely a must for advanced AEG systems.
- Directed exploitable heap layout derivation. Automatic exploitable heap layout derivation is a critical problem for AEG systems when tackling heap-related vulnerabilities including heap overflow, use-after-free (abbr. UAF), double-free, etc., as exploitation of these vulnerabilities generally requires a specific POC input that can lead the program to reach an exact heap region layout. Though several works including RELAY [31], HAEPG [32], etc., have proposed some passive pattern-matching-based solutions selecting out the execution states met with the prerequisite of some exploitable pattern and then performing an AEG process upon them, directed automatic exploitable heap layout deduction, however, we believe, is an important researching topic that can enhance the efficiency in exploitable heap interaction sequence generation.
- Adaption onto multi-step exploitation automaton. Currently, ExpGen implements a two-step exploitation automaton, one step for address leakage, the other control flow hijack. This can, however, be easily adapted to multi-step exploitation automaton, upon which we may be able to evaluate the compositional exploitability with multiple vulnerabilities on hand.

9. Related Work

9.1. Feedback-Directed Fuzzing

Since AFL [5] was proposed, feedback-directed fuzzing has become the most promising fuzzing technique. Variants including honggfuzz [33], AFL++ [17] were proposed. These fuzzers generally instrument binary programs through dynamic binary translation [34] or dynamic binary instrumentation [35], then collect the runtime branch coverage information during executions of the generated test cases. If a test case can trigger some new coverage that has not been seen before, this test case is viewed as interesting, and the fuzzer would later generate new test cases upon it.

The branch coverage is the original type of feedback a fuzzer collects from the target program during the fuzzing stage. Motivated by this example, researchers have defined many other types of feedback information and use these information to construct an advanced evolutionary fuzzer. IJON [36] provides several annotation techniques for analysts to directly add customized feedback mechanisms onto the source code of the analyzed programs. Angora [37] tracks the taint information alongside the execution to ncrease branch coverage by solving path constraints without symbolic execution. GREYONE [38] performs data-flow-sensitive fuzzing. It infers taint of variables through dynamic monitoring, and, based on this information, dynamically makes decisions on which branch to explore, which bytes to mutate and how to mutate in the fuzzing process. UAFL [39] introduces a new type of feedback called 'typestate' and devises a typestate-guided fuzzer to discover use-after-free vulnerabilities from the target programs.

9.2. Automatic Exploit Generation

Brumley et al. [40] introduced an automatic patch-based solution that can generate exploit input for programs where patches for some vulnerabilities are available. Though effective in five Microsoft programs, its exploitation automaton is only able to cover a small fraction of vulnerability exploitation cases. In 2011, Thanassis Avgerinos et al. proposed AEG [41]. AEG is, to our knowledge, the first end-to-end system for automatic exploit generation. By implementing preconditioned symbolic execution and targeting symbolic execution, it can automatically generate exploits for control-flow hijacking vulnerabilities. One year later, MAYHEM [14] was proposed, enhancing AEG in hybrid symbolic execution and index-based memory modeling. Huang et al. proposed CRAX [12]. Based on the open-source binary symbolic execution engine S2E [18], CRAX dynamically monitors a POC input for a specific vulnerability in a concolic execution way. By collecting the path constraint and crafting the exploitable constraint, CRAX is able to generate exploits for vulnerabilities including format string, stack overflow, etc.

Automatic exploit generation for heap-related vulnerabilities has also become a hot research topic nowadays. Sean Heelan et al. proposed SHRIKE [42], a pseudo-random black-box search algorithm that searches for the inputs required to place the source of a heap-based buffer overflow or underflow next to heap-allocated objects that an exploit developer, or automatic exploit-generation system, wishes to read or corrupt. However, SHRIKE is only applicable to PHP programs. For binary programs, Deng et al. proposed a pattern-based exploitability evaluation framework RELAY [31] capable of generating an exploit input to evaluate metadata corruption vulnerabilities. Wang et al. proposed revery [43], a system able to generate exploits on heap-related vulnerabilities with POC input that cannot lead to the exploitable state provided. A novel layout-oriented fuzzing technique and a control-flow stitching solution were also introduced in revery. Based on revery, maze [44] was also proposed, modeling the heap-layouts techniques (such as heap feng shui), and implementing automated heap-layout manipulation. Zhao et al. proposed propose HAEPG [32], an automatic exploit framework that can utilize known exploitation techniques to guide exploit generation.

However, none of the above AEG solutions make separate discussions and studies on vulnerability exploitation in an ASLR environment. Austin Gadient proposed Marten [25], which is, to our knowledge, the first attempt on this issue. However, as mentioned in

Section 2.2.2, we find the relocation Marten performs on an analysis-time exploit is only in a literal form, ignoring the complex semantic information imposed by the execution of the target program; therefore it would fail in circumstances where the execution of a program imposes complex constraint on the to-be-relocated parts of the analysis-time exploit. Our method, however, because of the semantic aware runtime exploit relocation technique it implements, can tackle this issue properly, providing solution much more complete than Marten's.

10. Conclusions

In this paper, we propose a novel two-step vulnerability-exploitability evaluation solution ExpGen for binary programs running in an environment where ASLR is deployed. We devise a partial-exploit sensitive POC-generation mechanism and exploitation context sensitive analysis-time exploit generation scheme. These two techniques cooperate together to generate an exploit input that can exercise all the necessary steps desired to tackle the issue on vulnerability exploitation in an analysis-time ASLR environment. Then, through the runtime exploit relocation technique we proposed, the gap between analysis-time address-space layout and runtime address-space layout is fixed, providing a completely automated exploitability-evaluation pipeline. Experimental results demonstrate that ExpGen has advantages over current AEG solutions in both address-leakage exploit generation and runtime exploitation session adaption.

Author Contributions: Conceptualization, H.H. and Y.L.; methodology, H.H. and Y.L.; software, H.H. and Z.P.; validation, H.H.; investigation, H.H.; resources, Y.L. and Z.P.; writing-original draft preparation, H.H.; writing-review and editing, H.H., Y.L., Z.P., K.Z., L.Y. and L.Z.; supervision, Y.L. and Z.P.; project administration, Y.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by National Key Research and Development Project of China (No. 2017YFB0802900).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: We would like to sincerely thank all the reviewers for your time and expertise on this paper. Your insightful comments help us improve this work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Sutton, M; Greene.A; Amini, P. *Fuzzing: Brute Force Vulnerability Discovery*; Addison-Wesley Professional: Boston, MA, USA, 2007.
2. Manès, V.J.M.; Han, H.; Han, C.; Cha, S.K.; Egele, M.; Schwartz, E.J.; Woo, M. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2312–2331.
3. King, J.C. Symbolic Execution and Program Testing. *Commun. ACM* **1976**, *19*, 385–394. https://doi.org/10.1145/360248.360252.
4. Cadar, C.; Dunbar, D.; Engler, D.R. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In Proceedings of the 8th USENIX Conf. on Operating Systems Design and Implementation, USENIX Association, 2008, OSDI'08, San Diego, CA, USA, 8–10 December 2008; pp. 209–224.
5. American Fuzzy Lop. Available online: https://lcamtuf.coredump.cx/afl/ (accessed on 21 April 2022).
6. Aschermann, C.; Schumilo, S.; Blazytko, T.; Gawlik, R.; Holz, T. REDQUEEN: Fuzzing with Input-to-State Correspondence. In Proceedings of the Symposium on Network and Distributed System Security (NDSS), San Diego, CA, USA, 24–27 February 2019.
7. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In Proceedings of the Proceedings 2016 Network and Distributed System Security Symposium, San Diego, CA, USA, 21–24 February 2016. https://doi.org/10.14722/ndss.2016.23368.
8. Yun, I.; Lee, S.; Xu, M.; Jang, Y.; Kim, T. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 745–761.
9. !Exploitable Crash Analyzer Version 1.6. Available online: https://www.microsoft.com/security/blog/2013/06/13/exploitable-crash-analyzer-version-1-6/ (accessed on 23 April 2022).

10. He, L.; Cai, Y.; Hu, H.; Su, P.; Liang, Z.; Yang, Y.; Huang, H.; Yan, J.; Jia, X.; Feng, D. Automatically assessing crashes from heap overflows. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, 30 October–3 November 2017; pp. 274–279. https://doi.org/10.1109/ASE.2017.8115640.
11. Cyber Grand Challenge (CGC) (Archived). Available online: https://www.darpa.mil/program/cyber-grand-challenge (accessed on 22 April 2022).
12. Huang, S.K.; Huang, M.H.; Huang, P.Y.; Lai, C.W.; Lu, H.L.; Leong, W.M. 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, Aithersburg, MD, USA, 20–22 June 2012; pp. 78–87. https://doi.org/10.1109/SERE.2012.20.
13. Shellphish. Rex. Available online: https://github.com/angr/rex (accessed on 23 April 2022).
14. Cha, S.K.; Avgerinos, T.; Rebert, A.; Brumley, D. Unleashing Mayhem on Binary Code. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 380–394. https://doi.org/10.1109/SP.2012.31.
15. Barrett, C.W.; Sebastiani, R.; Seshia, S.A.; Tinelli, C. Satisfiability Modulo Theories. *Handb. Satisf.* **2009**, *185*, 825–885.
16. Address Space Layout Randomization. Available online: https://en.wikipedia.org/wiki/Address_space_layout_randomization (accessed on 22 April 2020).
17. Fioraldi, A.; Maier, D.; Eißfeldt, H.; Heuse, M. AFL++: Combining Incremental Steps of Fuzzing Research. In Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association, Boston, MA, USA, 10–11 August 2020.
18. Chipounov, V.; Kuznetsov, V.; Candea, G. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst. (TOCS)* **2012**, *30*, 2:1–2:49. https://doi.org/10.1145/2110356.2110358.
19. Position-Independent Code. Available online: https://en.wikipedia.org/wiki/Position-independent_code (accessed on 22 April 2020).
20. Notes on Non-Executable Stack. Available online: https://web.ecs.syr.edu/~wedu/seed/Labs_12.04/Files/NX.pdf (accessed on 22 April 2020).
21. Huang, N.; Huang, S.G.; Deng, Z.K. Automatic Detection of Stack Overflow Attack in Canary. In Proceedings of the 8th International Conference on Instrumentation and Measurement, Computer, Communication and Control (IMCCC), Harbin, China, 19–21 July 2018; pp. 1418–1423.
22. Serebryany, K.; Bruening, D.; Potapenko, A.; Vyukov, D. AddressSanitizer: A Fast Address Sanity Checker. In Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), Berkeley, CA, USA, 13–15 June 2012; USENIX Association: Boston, MA, USA, 2012; pp. 309–318.
23. Return-to-Libc Attack. Available online: https://en.wikipedia.org/wiki/Return-to-libc_attack (accessed on 22 April 2020).
24. ret2text. Available online: https://www.ctfnote.com/pwn/linux-exploitation/rop/ret2text (accessed on 22 April 2020).
25. Gadient, A.J. Automated Exploitation of Fully Randomized Executables. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2019.
26. Global Offset Table (Processor-Specific). Available online: https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/6n33n7fe3/index.html (accessed on 11 June 2022).
27. Pwntools Github Repository. Available online: https://github.com/Gallopsled/pwntools (accessed on 22 April 2022).
28. De Moura, L.; Bjørner, N. Z3: An efficient SMT solver. In Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary, 29 March–6April 2008; pp. 337–340.
29. CQGameRHG2021ljb. Available online:https://github.com/chunqiugame/CQGame_RHG_2021ljb (accessed on 22 April 2020).
30. CQGameRHG. Available online: https://github.com/chunqiugame/CQGame_RHG (accessed on 22 April 2020).
31. Deng, F.; Wang, J.; Zhang, B.; Feng, C.; Jiang, Z.; Su, Y. A Pattern-Based Software Testing Framework for Exploitability Evaluation of Metadata Corruption Vulnerabilities. *Sci. Program.* **2020**, *2020*, 8883746. https://doi.org/10.1155/2020/8883746.
32. Zhao, Z.; Wang, Y.; Gong, X. HAEPG: An Automatic Multi-hop Exploitation Generation Framework. In *Detection of Intrusions and Malware, and Vulnerability Assessment*; Maurice, C., Bilge, L., Stringhini, G., Neves, N., Eds.; Springer: Berlin/Heidelberg, Germany, 2020; Volume 12223, pp. 89–109, https://doi.org/10.1007/978-3-030-52683-2_5.
33. Honggfuzz. Available online: https://github.com/google/honggfuzz (accessed on 21 April 2020).
34. Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, Anaheim, CA, USA, 10–15 April 2005; pp. 41–46.
35. Pin—A Dynamic Binary Instrumentation Tool. Available online: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html (accessed on 22 April 2020).
36. Aschermann, C.; Schumilo, S.; Abbasi, A.; Holz, T. IJON: Exploring Deep State Spaces via Fuzzing. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; p. 16.
37. Chen, P.; Chen, H. Angora: Efficient Fuzzing by Principled Search. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 711–725. https://doi.org/10.1109/SP.2018.00046.
38. Gan, S.; Zhang, C.; Chen, P.; Zhao, B.; Qin, X.; Wu, D.; Chen, Z. GREYONE: Data Flow Sensitive Fuzzing. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, Boston, MA, USA, 12–14 August 2020; pp. 2577–2594.
39. Wang, H.; Xie, X.; Li, Y.; Wen, C.; Li, Y.; Liu, Y.; Qin, S.; Chen, H.; Sui, Y. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Korea, 5–11 October 2020; pp. 999–1010. https://doi.org/10.1145/3377811.3380386.

40. Brumley, D.; Poosankam, P.; Song, D.; Zheng, J. Automatic patch-based exploit generation is possible: Techniques and implications. In Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008), Oakland, CA, USA, 18–22 May 2008; pp. 143–157.
41. Avgerinos, T.; Cha, S.K.; Hao, B.L.T.; Brumley, D. AEG: Automatic Exploit Generation. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 6–9 February 2011.
42. Heelan, S.; Melham, T.; Kroening, D. Automatic Heap Layout Manipulation for Exploitation. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), USENIX Association, Baltimore, MD, USA, 14–16 August 2018; pp. 763–779.
43. Wang, Y.; Zhang, C.; Xiang, X.; Zhao, Z.; Li, W.; Gong, X.; Liu, B.; Chen, K.; Zou, W. Revery: From Proof-of-Concept to Exploitable. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 1914–1927. https://doi.org/10.1145/3243734.3243847.
44. Wang, Y.; Zhang, C.; Zhao, Z.; Zhang, B.; Gong, X.; Zou, W. MAZE: Towards Automated Heap Feng Shui. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), USENIX Association, virtual, 11–13 August 2021; pp. 1647–1664.