HAEPG: An Automatic Multi-hop Exploitation Generation Framework

Zixuan Zhao[1,2,3,4], Yan Wang[1,2,3,4], and Xiaorui Gong[1,2,3,4(✉)]

[1] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[2] Key Laboratory of Network Assessment Technology, Beijing, China
[3] Beijing Key Laboratory of Network Security and Protection Technology, Beijing, China
[4] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{zhaozixuan,wangyan,gongxiaorui}@iie.ac.cn

Abstract. Automatic exploit generation for heap vulnerabilities is an open challenge. Current studies require a sensitive pointer on the heap to hijack the control flow and pay little attention to vulnerabilities with limited capabilities. In this paper, we propose `HAEPG`, an automatic exploit framework that can utilize known exploitation techniques to guide exploit generation. We implemented a prototype of `HAEPG` based on the symbolic execution engine S2E [15] and provided four exploitation techniques for it as prior knowledge. `HAEPG` takes crashing inputs, programs, and prior knowledge as input, and generates exploits for vulnerabilities with limited capabilities by abusing heap allocator's internal functionalities.

We evaluated `HAEPG` with 24 CTF programs, and the results show that `HAEPG` is able to accurately reason about the type of vulnerability for 21 (87.5%) of them, and generate exploits that spawn a shell for 16 (66.7%) of them. All the exploits could bypass NX [25] and Full RELRO [28] security mechanisms.

Keywords: Automatic exploit generation · Heap vulnerability · Symbolic execution

1 Introduction

Automated exploit generation (AEG) is becoming an important method in vulnerability-centric attacks and defenses. Software vendors use it to evaluate the severity of security vulnerabilities more quickly and allocate appropriate resources to fix critical vulnerabilities. Defenders learn from synthetic exploits to generate Intrusion Detection System rules and block potential attacks.

Most AEG solutions [12,13,20,23,26] usually only support stack-related or format string vulnerabilities, which are rare in modern systems [2]. Due to the complexity of heap allocator functions, only a few existing solutions can generate exploits for heap-based vulnerabilities. These solutions have different approaches.

C. Maurice et al. (Eds.): DIMVA 2020, LNCS 12223, pp. 89–109, 2020.
https://doi.org/10.1007/978-3-030-52683-2_5

For instance, Revery [30] applies a layout-oriented fuzzing and control-flow stitching solution to explore exploitable states in paths derived from vulnerability points. Gollum [22] employs a custom heap allocator to create exploitable heap layouts and a fuzzing technique based on prior work [21] to solve the heap manipulation problem. SLAKE [14] uses a static-dynamic hybrid analysis to search for useful kernel objects and manipulates heap layout by adjusting the free list in the slab.

All these solutions corrupt a sensitive pointer (e.g., VTable pointer) and derive an attacker-controlled memory-write or indirect call, which means that the presence of a sensitive pointer is key to hijack the control flow. In this case, once the heap layout is well arranged, an attacker creates an exploit primitive with only one operation, i.e., triggering the vulnerability, and we call it single-hop exploitation. However, not all vulnerabilities can be exploited using simple single-hop techniques. For example, with an off-by-one error [11], it is infeasible to fully control any sensitive pointer by merely triggering the vulnerability, let alone overwriting the instruction pointer to an arbitrary value. To solve this issue, the following challenges need to be addressed:

Challenge 1: Exploring the Power of Heap Vulnerabilities with Limited Capabilities. To exploit vulnerabilities with limited capabilities, an attacker needs to manipulate the heap layout and abuse the heap allocator's internal functionalities to create several intermediate hops, expand the range of corruptible memory with the help of the hops, and eventually derive an arbitrary memory-write or indirect call. We call these techniques multi-hop exploitation. Some solutions [19,32] aim to discover such techniques for heap allocators. However, they can not apply the techniques to programs with heap-based vulnerabilities automatically. To the best of our knowledge, existing AEG solutions paid very little attention to it.

Challenge 2: Modeling Heap Interactions Between Programs and Heap Allocators. To conduct multi-hop exploitation, AEG solutions have to craft inputs and drive victim programs to allocate and deallocate objects of a specific size or write specific data to heap objects. However, programs typically do not expose any direct interfaces for users to interact with their heap allocators. Therefore, AEG solutions have to recognize heap interactions and assemble them in a particular way to generate exploits.

Our Solution. In this paper, we propose `HAEPG` to address the challenges above. Given a program with heap-based vulnerabilities and crashing inputs, it attempts to achieve the execution of arbitrary code through multi-hop exploitation.

`HAEPG` abstracts machine-level instructions and function calls interacting with the heap allocator as heap interactions. It relies on the fact that most programs distribute functions with function dispatchers (e.g., event handling and connection processing loops) and extracts the paths that make up such dispatchers. Then, `HAEPG` applies hybrid techniques to locate and analyze heap interactions and infer dependencies between different interactions and paths.

After this, HAEPG collects runtime information in programs when executing crashing inputs. It inspects vulnerable objects and analyzes the type of memory corruption as well as the size of corrupted data.

Furthermore, we studied manual multi-hop exploitation techniques for heap vulnerabilities. These techniques usually abuse the heap allocator's internal functionalities and improve the vulnerabilities' capability by carefully crafted heap interaction sequences. We designed a templating language to abstract known multi-hop exploitation techniques as exploit templates. HAEPG uses them to achieve an arbitrary execution and generate end-to-end exploits.

We built a prototype of HAEPG based on the symbolic execution engine S2E [15] and wrote templates for four exploitation techniques of ptmalloc [4], the standard allocator of glibc, and evaluated it on 24 programs from well known Capture The Flag (CTF) competitions. The results show that HAEPG is able to accurately reason about the type of vulnerability for 21 (87.5%) of them, and generate exploits that spawn a shell for 16 (66.7%) of them.

2 Motivational Example

In this section, we give an example to illustrate multi-hop exploitation and reveal problems AEG solutions encounter when handling the example.

The Vulnerability. The example is running on a GNU/Linux system with an unmodified version of glibc.. As shown in Fig. 1, the program has three functions, i.e., $addItem$, $removeItem$, $editItem$, which are used to allocate an object, release an object, and modify an object. There is a poison-null-byte error [16] at Line 22, but it only corrupts the meta-data between heap objects, while the content of the heap objects remains unaltered.

Multi-hop Exploitation. The example in Fig. 1 shows the exploitation via the unsafe unlink technique [9]. We first allocate three heap objects A, B, and C. The pointer that the program used to access object B is stored in BSS. Then, we trigger the vulnerability in object A to shrink the object B's size, as shown in *state 3*, and forge a fake chunk in object A in *state 4*. The fake chunk is well arranged to bypass sanity checks and leads to an arbitrary write primitive in *state 6* after releasing object B in *state 5*. Finally, we corrupt a function pointer with the arbitrary write primitive and hijack the control flow.

These states can be categorized as follows:

- **Initial state:** State when the program starts running, e.g., *state 1*.
- **Preparation state:** State when the program manipulates memory layouts for exploitation before the corruption happens, e.g., *state 2*.
- **Corrupting state:** State when triggering the vulnerability, e.g., *state 3*.
- **Intermediate state:** State that the program would go through for reaching an exploitable state from the initial state, e.g., *state 4-5*.
- **Exploitable state:** State with an exploit primitive for exploitation, e.g., *state 6* and *7*.

```
1. void addItem(){
2.     int size = read_int();
3.     size_list[index] = size;
4.     heap_list[index] = malloc(size);
5.     if(!heap_list[index])
6.         puts("malloc error");
7.     return;
8. }
9. void removeItem(){
10.    if(heap_list[index]){
11.        free(heap_list[index]);
12.        heap_list[index] = 0;
13.    }
14.    if(size_list[index])
15.        size_list[index] = -1;
16. }
17. void editItem(){
18.    for(int i = 0; i < size_list[index]; i++){
19.        read(0, heap_list[index] + i, 1);
20.        if(!heap_list[index][i])
21.            break;
22.    }
23.    heap_list[index][size_list[index]] = NULL;
24. }
25. void main(){
26.    while(True){
27.        index = readInst();
28.        choice = readInst();
29.        switch(choice){
30.            case 1: addItem(); break;
31.            case 2: removeItem(); break;
32.            case 3: editItem(); break;
33.        }
34.    }
35. }
```

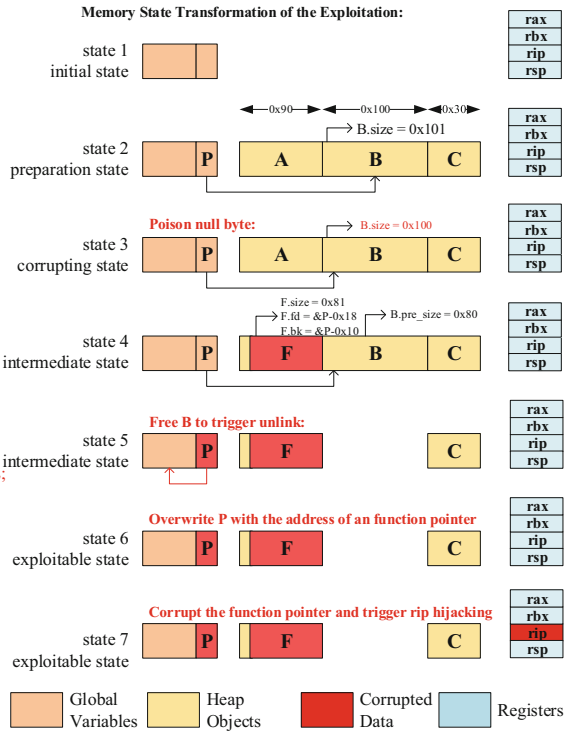

Fig. 1. An example of poison-null-byte

It is easy for modern fuzzing tools [10,33] to generate crashing inputs for the vulnerability. However, most of the AEG solutions could not handle this case because there is no direct exploit primitive upon crash of the program. For instance, the auto-exploit kit framework Mechanical Phish [26], which is developed by Shellphish and came third in DARPA CGC [17], could only detect the vulnerability and generate no exploit for the example, because Mechanical Phish requires a controllable pointer for injecting shellcodes or rop-chains. The solution Revery [30] could find the corrupting *state 3*. However, it has no capabilities for bypassing the heap allocator's sanity checks and enhancing the vulnerability's capability, and thus could not turn the vulnerability into an exploit.

3 Methodology

Figure 2 shows an overview of `HAEPG`. It takes programs and crashing inputs as input, and templates for the guidance of exploitation. `HAEPG` first models heap interactions of the target program with function paths and heap primitives. It extracts function paths from the program using static analysis, and dynamically

tracks instructions and function calls of the function paths interacting with the heap allocator during runtime and records relations of different heap interactions.

Then, HAEPG runs the program with the crashing input to retrieve information about the vulnerability, including its type and the scale of corrupted data. We designed a templating language for templating widely used exploitation techniques. Each template contains the necessary information for guiding the exploitation. HAEPG filters applicable exploit templates by checking if the heap allocator and the program meets the templates' requirements. It attempts to generate an attack sequence and cyclically assesses and corrects the sequence until the program reaches an exploitable state, or the analysis exceeds a configurable upper bound for generation attempts (e.g., 20 times).

Finally, HAEPG uses the generated exploit primitive to corrupt the instruction pointer for transferring control flow to one-gadget [18]. One-gadgets are code fragments inside glibc that invokes "/bin/sh" without any arguments, effectively spawning a shell for the attacker. Once HAEPG detects a shell process is created in the target program, it solves the path and data constraints collected when executing the attack sequence and generates an exploit input.

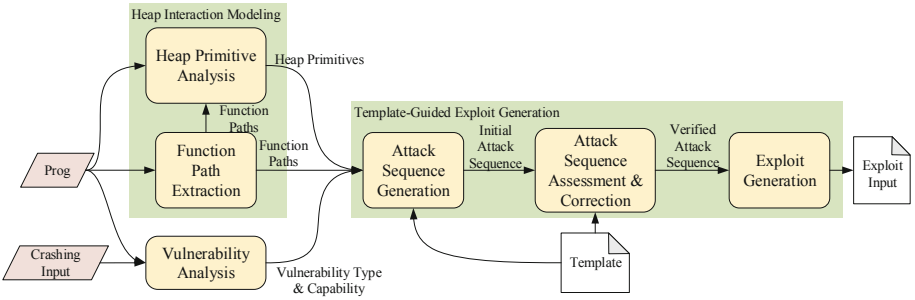

Fig. 2. Overview of HAEPG

3.1 Heap Interaction Modeling

Function Path Extraction. A function dispatcher is a code structure that is widely used in programs for function distribution, and usually implemented as if-else or switch-case structure wrapped in a loop. Programs cyclically receive instructions at the loop entry and execute corresponding functions. We define the basic block sequences from the loop entry to the loop exit as function paths, which have the following properties:

– **Atomicity:** Function paths are indivisible at runtime. A program cannot execute half of a function path and then jump to other function paths. The structure of the function dispatcher has determined that each function path must be executed entirely before executing others.

– **Reentrant:** Function paths have the same entrance and exit. When it reaches the exit of the function path, the program returns to the entrance and chooses the next function path according to instructions. Thus a program can execute a function path several times through proper instructions.

To extract function paths, `HAEPG` first generates a control-flow graph of the target program and marks the loops containing the function calls of heap allocations or deallocations as candidate function dispatchers. If the loop body of the function dispatcher is a switch-case statement, `HAEPG` will search for the basic block with several successors and extract the path of each successor as a function path. Otherwise, the loop body could be a sequence of nested if-else statements. `HAEPG` will traverse the loop backward the starting from the end of the loop body and check the number of each basic block's precedents during the traversal. The first basic block, which has a large number of precedents, e.g., more than 5, would be marked as the merge point of all dispatched functions, and we extract all paths from the loop entry to this block as function paths.

Heap Primitive Analysis. Heap primitives model the interactions between programs and heap allocators. We distinguish between the following three types and their attributes:[1]

Allocation	**Deallocation**	**Edit**
size: the size of allocation	*addr*: the address to be released	*base*: the base of edit address
addr: the address returned by the heap allocator		*offset*: the offset of edit address
		data: the data to be written

In general, heap primitives are these program machine instructions: (1) function calls interacting with the heap allocator, such as malloc, calloc, free, etc. (2) function calls with heap pointers as arguments, such as read, fgets, etc. (3) memory-write instructions with heap objects as the destination address.

To analyze heap primitives, `HAEPG` executes function paths using symbolic execution. It symbolizes all input bytes and tracks instructions and the calling of APIs interacting with the heap region. `HAEPG` only records instructions and the calling of the APIs in the target program as heap primitives and ignores shared libraries, because tracking both the target program and the shared libraries would increase performance overhead. If any attribute of heap primitives are symbolic, `HAEPG` will solve and record the range of it without concretizing it. To reason about the relationship between heap primitives, `HAEPG` associates a globally unique taint tag for each heap pointer returned by an allocation and identifies primitives that operated on a tainted pointer.

3.2 Vulnerability Analysis

We obtain crashing inputs with AFL [33] and analyze the vulnerability's capability by detecting violations of memory usage. `HAEPG` dynamically executes the

[1] Note that we only model the basic interaction types. Heap allocators can have other types of interactions (e.g., realloc), which are outside the scope of this paper.

crashing inputs and taints return pointer of allocations similarly to the heap primitive analysis. Additionally, it propagates the tag to the pointed memory object's bytes. We further add annotations about a heap object's status to the tags: Initially, tags are `uninitialized`, and instructions writing to the object will change the status to `busy` while instructions releasing the object will change it to `free`. Furthermore, `HAEPG` records the size of corresponding objects for tags. If any instruction accesses the heap memory, we could get the pointer's tag `ptag` and the pointed object's memory tag `mtag`, together with the statuses and sizes of them. Before changing the statuses of them, `HAEPG` checks if any of the violation rules shown in Table 1 is triggered.

Table 1. List of vulnerability types, trigger options, and violation rules

Vulnerability type	Trigger operation	Violation rule
Double free	Free a heap chunk	`mtag.status` == `free` or `ptag.status` == `free`
Use After Free	Store `n` bytes of `data` in memory address [`base` + `off`] (`base` and `off` are the base and offset of addressing)	`mtag.status` == `free` or `ptag.status` == `free`
Overflow		`n` + `off` >`ptag.size`
Poison Null Byte		`n` + `off` == `ptag.size` + 1 and the last byte of `data` is null byte
Off by One		`n` + `off` == `ptag.size` + 1

If a violation rule is triggered, `HAEPG` will record the scale of corrupted data, such as the range of overflowed bytes or the size of the vulnerable chunk.

3.3 Template-Guided Exploit Generation

In this section, we will illustrate our method of exploit generation. We bring existing exploitation techniques as prior knowledge of constructing memory states and reaching exploitable states into `HAEPG`. Moreover, instead of hard-coding exploitation techniques into `HAEPG`, we developed a templating language to describe exploitation techniques. The method of dynamic interpreting templates and constructing attack sequences provides flexibility and extendability to `HAEPG`.

Templates. The procedure of applying exploitation techniques is program-sensitive, as even slight changes in the target program result in need of vastly different exploitation strategies. Thus we collect constant and essential components of exploitation techniques and abstract them as templates, which give information over the following components:

- **Backbone Primitives Sequence:** The order of heap primitives, which used to trigger the vulnerability and abuse internal functionalities of heap allocators, remains constant. For example, cyclically releasing a heap chunk in the fastbin attack to gain an arbitrary allocation from a double-free vulnerability [5]. We refer to such heap primitives as a backbone primitives sequence.
- **Layout Constraints:** Intermediate states of the multi-hop exploitation may need to meet certain constraints. For example, an unsafe unlink attack requires the victim chunk to be allocated in unsortedbin size, and the fake chunk to be adjacent to the victim chunk. We refer to such constraints as layout constraints.
- **Requirements:** To use an exploitation technique, the program has to meet some requirements. For example, the program must have the ability to allocate objects in fastbin size for fastbin attack.

These components indicate how to construct memory states to reach an exploitable state in multi-hop exploitation and constraints that memory states should satisfy. Likewise, each template consists of three parts, including requirements of using the template, backbone primitives, and layout constraints. We will introduce the templating language we used to abstract exploitation techniques in Sect. 4.3.

Attack Sequence Generation. Attack sequence generation is closely tied to the provided templates. Firstly, `HAEPG` checks if the target program meets the specified requirements, such as the vulnerability type and the glibc version. If the program passed these checks, `HAEPG` will select this template for the next steps; otherwise, it will try other templates.

Next, `HAEPG` traverses the backbone primitives of the template. It scans all function paths to find the ones containing the backbone primitives, and combines these function paths together with their heap primitives as an attack sequence.

We designed the following two methods to execute an attack sequence:

- **Heap Simulator:** The simulator is an independent binary that uses the same heap allocator as the target program (i.e., ptmalloc for our prototype). We use it to execute the heap primitives of the attack sequence and simulate the intermediate states of the target program.
- **Symbolic Execution:** We let the target program execute function paths of the attack sequence and associate heap primitives as data constraints with the interrelated memory data and registers in S2E (e.g., transform the size of allocations to data constraints of malloc/calloc's first argument).

Attack Sequence Assessment and Correction. `HAEPG` assesses the attack sequence according to the layout constraints of the template. It dynamically executes the target program and the attack sequence and records heap layouts at runtime. Then, it extracts the address of each essential heap object, the real size of them, and the address where the object pointers are stored, etc. With this information, once precedent backbone primitives of a memory state are finished,

HAEPG will check if the current heap layout meets the layout constraints, including the distance of heap objects and the status of them.

To improve performance, HAEPG first uses the heap simulator for a quick assessment. Only after the attack sequence passed the assessment, HAEPG will use S2E for an accurate assessment.

If the layout constraints are met, the program will enter an exploitable state, and HAEPG would attempt to generate an exploit. Otherwise, HAEPG will find the conflicting heap layout. Then, HAEPG will infer the reason for the conflict and attempt to correct it. In general, the reasons for the conflict are as follows:

1. Heap chunk A should be `free` but it is `busy` or `uninitialized`;
2. Heap chunk A and B must be adjacent but there are other `busy` chunks between them;
3. Heap chunk A and B must be adjacent but the adjacent chunk of A is `free`.

For the first case, HAEPG inserts a path into the attack sequence to release the chunk A. For the second case, HAEPG tries to insert paths to release all chunks in the middle of chunk A and B. If chunks cannot be released, HAEPG would allocate a heap chunk with the same size as chunk A before the allocation of it to get a new heap layout, which might lead to a satisfying heap layout. The third case is generally caused by extra heap chunks of the freelist which make chunk B gets allocated on a wrong slot, and HAEPG fills extra chunks by allocating chunks with the same size as B. Finally, HAEPG generates a new attack sequence and repeats the assessment and correction until it finds an attack sequence that leads to an exploitable memory state (i.e., a state matching the layout constraints).

Exploit Generation. Once the attack sequence passed the assessment, HAEPG executes it using symbolic execution and detect exploit primitives by tracking instructions of the target program and checking if symbolic data is present in one of the following locations: (1) the content and target address for memory-write instructions and function calls; (2) the head pointer of one of the bins; (3) the target address for indirect calls/jumps; (4) the value of function pointers of the program and glibc (e.g., GOT and *malloc_hook*).

Based on the location of symbolic data, HAEPG can use one of the following exploit primitives to derive an exploit input:

- **Arbitrary Execution (AX):** If the target address of indirect calls is symbolic, or any function pointer is symbolic, HAEPG could corrupt the instruction pointer to an arbitrary value. In this case, HAEPG uses a one-gadget [18] as the target address. Note that each one-gadget has individual memory and register constraints that need to be satisfied. Hence, HAEPG will check the related memory and registers when the instruction pointer is corrupted and pick a proper one-gadget for exploitation.
- **Arbitrary Write (AW):** If the value and the target address of any memory-write instructions or function calls are symbolic, HAEPG can write arbitrary data to an arbitrary location. HAEPG leverages it to overwrite a function pointer of the GOT or glibc (glibc is preferred if Full RELRO [28] is enabled)

and exercises a function path that calls the corrupted function pointer, which transforms the exploit primitive to an arbitrary execution.
- **Arbitrary Allocation (AA):** If the head pointer of a bin is symbolic, HAEPG can allocate a chunk at an arbitrary location. However, this primitive has different constraints based on the type of bin. For tcache, the allocator does not check the meta-data of the allocated chunk, so we treat the primitive as an arbitrary write primitive; For fastbin, the allocator checks the consistency of chunk size, so an attacker has to find or construct a fake meta-data to bypass the sanity check. Fortunately, there are some data in glibc which is suitable for bypassing the sanity check and transforming the exploit primitive to an arbitrary execution[2].

After corrupting the instruction pointer with a one-gadget, HAEPG will hook the APIs for process generation (e.g. execve). Once HAEPG detects an invokation of these APIs with the argument of bash's path, it solves constraints collected when executing the attack sequence and generates an exploit input.

4 Implementation

4.1 Static Analysis

Control Flow Graph Construction. As discussed by Shoshitaishvili et al. [27], it is very challenging to recover an accurate CFG for programs due to indirect calls. Since CFG generation is not a contribution of this paper, we only focus on programs with no recursion and indirect calls. We developed a simple CFG generation program for HAEPG's prototype, which constructs the CFG by extracting jump targets of basic blocks in the disassembly using static analysis. It is sufficient for our evaluation set.

Redundant Function Path Simplification. Since a function path is a sequence of basic blocks, branches create a new function path for each branch target. As we use dynamic analysis to infer the dependencies between function paths, a large number of function paths increases HAEPG's performance overhead. As we only focus on heap interactions, most of the function paths are redundant for HAEPG because heap interactions of them are the same or similar, which will cause dynamic analysis to do repetitive work. Besides, some function paths should be dropped because they are not related to exploitation. We simplify function paths with the following methods:

- **Merging:** We construct the CFG for the branch in Line 14 to Line 16 in Fig. 1, as shown in Fig. 3.a. To reduce the impact of branches on the number of paths, we extract an API call sequence from the paths and mark the

[2] For example, the function pointers nearby *malloc_hook* in glibc, which could be mistaken as valid meta-data by the allocator. We could directly overwrite *malloc_hook* and hijack the instruction pointer by allocating a chunk on it.

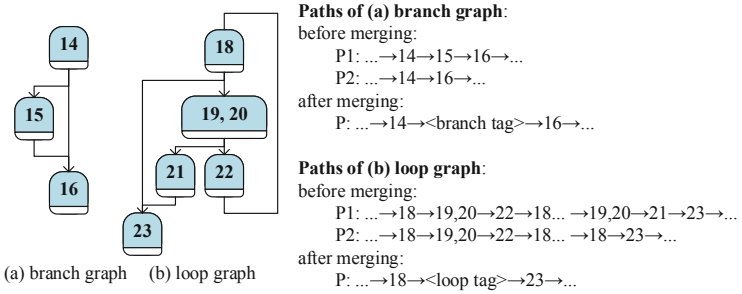

Fig. 3. Two types of structure to be optimized

branch with a `branch tag` if the API call sequence does not contain heap interactions.

A loop can have more than one exit, and the extra exits increase the number of function paths. We take the loop of $func3$ in Fig. 1 as an example, and the CFG is shown in Fig. 3.b. The loop has two exits, i.e., Line 21 and 23. Since Line 23 is the successor of Line 21, we regard Line 23 as the only exit of the loop and replace the loop with a ¡to merge function paths exiting from Line 21 or 23 into one.

– **Pruning:** Programs check functions' return values and perform different operations if they indicate the failure of function execution. Such operations are usually organized with conditional branches, which also increase the number of function paths. As exploiting the failure of function calls usually depends on programs rather than heap allocators, we consider them out of scope for `HAEPG`. We discard those paths as irrelevant and filter them out using a lightweight taint analysis for functions' return values.

4.2 Dependency of Function Paths

`HAEPG` may fail to execute function paths because program variables do not meet path constraints in symbolic execution. In this case, `HAEPG` executes other function paths first, which set the program variables correctly. We refer to these variables as reliant variables, and the latter function paths are the dominant paths of the formers. For example, we assume that function path $FP1$ is the sequence of line number 27-29-30-5-7 in Fig. 1, and $FP2$ is 27-29-31-10-11-14-15 (we only take the line number of branches). `HAEPG` would fail to jump to Line 11 from Line 10 when executing $FP2$ without executing $FP1$ first because the value of $heap\_list[index]$ does not meet $FP2$'s path constraints. In this case, $heap\_list[index]$ is the reliant variable of $FP2$, and $FP1$ is the dominant path of $FP2$.

`HAEPG` has to find all dominant paths for such function paths to avoid missing heap primitives in them. It first dynamically executes function paths and records the following information: (1) new values that the function path used to overwrite or update the original reliant values; (2) constraints not being satisfied and

Listing 1. Template of unsafe unlink

```
1  RQMT: Include(UNSORTEDBIN, hmodel.malloc_sizes) and
2        Include(vuln.type, ["Off-by-One", "Poison-Null-Byte", "Overflow"])
       ↪ and
3        VersionLower(allocator.ver, "2.26")
4  EXEC: vul_ptr = Allocation(size = (0x80 + RANDNUMB * 0x10 + 8), tag = vul.
       ↪ vul_tag)
5        vic_ptr = Allocation(size = (0xf0 + RANDNUMB * 0x100))
6        sep_ptr = Allocation(size = RANDNUMB)
7  SATS: adjacent(vul_ptr, vic_ptr) == True and adjacent (vic_ptr, topchunk)
       ↪ == False
8  EXEC: vul_data = h64(0) + h64(vul_ptr.size - 8 + 1) + h64(vul_ptr.base - 0
       ↪ x18) + h64(vul_ptr.base - 0x10) + RANDBYTE * (vul_ptr.size - 0x28)
       ↪  + h64(vic_ptr.size & (~8)) + "00"
9        Edit(base = vul_ptr, offset = 0, data = vul_data)
10 SATS: vul_ptr.chunk.fd == 0 and
11       vul_ptr.chunk.bk == (vul_ptr.chunk.raw_size - 0x10) and
12       vic_ptr.chunk.pre_size == (vic_ptr.chunk.bk - 1) and
13       vic_ptr.chunk.raw_size & 1 == 0
14 EXEC: Deallocation(vic_ptr)
15 EXEC: Edit(base = vul_ptr.base - 0x18, offset = 0, data = "A" * 0x20)
```

causing termination of states in symbolic execution. `HAEPG` infers expected values of reliant variables by solving these constraints and finds function paths that can set reliant variables correctly (i.e., the dominant paths).

4.3 Templating Engine

Templating Language. Our templating language describes exploitation techniques via requirements, backbone primitives, and layout constraints. We mark them in the template with labels *RQMT*, *EXEC*, and *SATS*, as shown in Listing 1. The language provides users with functions and objects to describe exploitation techniques, and Table 2 shows the central components of the language.

We employ the following methods to concretize the attributes of the heap primitives of attack sequences:

- **Direct Calculation:** `HAEPG` determines some of the attributes of heap primitives based on the range of them and the template, such as the size of an allocation. It solves these attributes when constructing the attack sequence. We refer to it as direct calculation.
- **Lazy Calculation:** Some of the attributes of heap primitives could only be determined at runtime, such as *vul_ptr.chunk*, *vul_ptr.base*, and the `data` of `edit` at Line 7 in Listing 1. They remain unsolved when the attack sequence is constructed. `HAEPG` collects runtime information and solves these attributes during the execution. We refer to it as lazy calculation.

Note that the `Allocation` function returns a *HeapChunk* object whose member variables represent meta-data's fields, the user payload, and the address

where the heap pointer is stored. A user does not need to initialize these member variables because HAEPG would automatically initialize them using lazy calculation. Listing 1 shows the template of unsafe unlink. It intuitively describes the exploitation of Fig. 1 showcased in.

When generating attack sequences, the backbone primitives from the same *EXEC* label are out of order, and HAEPG simulates the changes of reliant variables and sorts backbone primitives on the premise that reliant variables meet the expectation of function paths.

Heap Simulator. The main part of the simulator is a loop that wraps the three heap primitive functions described in Sect. 3.1. It takes heap primitives as the input and executes the corresponding functions to simulate the heap interactions of the target program. After finishing the execution of each function, the simulator will output its heap layout. HAEPG uses the simulator to simulate the memory states of the target program.

5 Evaluation

To evaluate the effectiveness of HAEPG, we implemented a prototype of HAEPG and assessed it with 24 programs CTF challenges, and all of them can be found in ctftime.org [1], pwnable.tw [7], and github.com [3]. We selected programs based on the following criteria: (1) programs must have at least one heap vulnerability, and vulnerabilities are diverse; (2) programs with higher scores are preferred. In general, challenges with higher scores are more difficult. Most selected challenges have higher scores than the median score for their CTF game, and 4 of them have the highest score in the exploitation category. We wrote templates for four common heap exploitation techniques: fastbin attack [5], unsafe unlink [9], house of force [6], and tcache poisoning [8], which are applicable to a significant amount of CTF challenges. However, some challenges are not shipped with their respective glibc, and we provided default glibc for them (2.27 for those whose intended solution involves tcache and 2.24 for others)[3]. The result shows that HAEPG can generate exploits for most of them.

All programs are tested in Ubuntu18.04, with Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz*24 and 512GB RAM. We enabled NX [25] and disabled ASLR [24] for all programs. We disabled ASLR because bypassing ASLR it is an orthogonal problem out of the scope of this paper.

5.1 Effectiveness

Table 3 shows the result of our evaluation. It contains details of programs, such as names and CTF competitions. Moreover, it shows the glibc version, the vulnerability type that HAEPG identified, and whether HAEPG could generate an exploit

[3] We provided ld.so for different versions of glibc, and changed the absolute paths of ld.so and libc.so to relative paths for all test cases. In this way, we were able to load arbitrary ld.so and libc.so on our evaluation system.

Table 2. List of function and variables provided by the templating language

Types	Name	Description
Object	vuln	vulnerability information, including vulnerability type and capability (for example, overflowed data size)
	hmodel	heap interaction model, including function paths and their heap primitives
	allocator	the allocator, including the version information
Type	HeapChunk	value returned by Allocation
Function	Include	check if two parameters are inclusive
	Allocation	allocation primitive
	Deallocation	deallocation primitive
	Edit	edit primitive
	Adjacent	check if two heap objects are adjacent in heap layout
	Distance	return the distance of two heap objects

for them. As a result, `HAEPG` accurately reasons about the type of vulnerability for 21 (87.5%) programs successfully and generate working exploits for 16 (66.7%) of them. Moreover, we bypassed Full RELRO [28] by corrupting the function pointers in glibc (e.g., *malloc_hook*) instead of GOT.

We also evaluated Revery [30] and Mechanical Phish [26] as a comparison, and none of them could generate exploits for these programs. Revery found corrupting states for these challenges. It reached unlink states for challenges that could be exploited with unsafe unlink (the results of them are marked with "*" in Table 3). However, Revery could not generate complete exploits because it uses fuzzing to explore the memory state space. Unfortunately, this is insufficient to forge fake chunks. Mechanical Phish found crashing states for these challenges with Driller [29], but it could not generate exploits because there was no pointer corrupted with symbolic bytes. Hence, Mechanical Phish could not hijack the instruction pointer or inject shellcodes/rop-chains.

5.2 Performance

To evaluate the performance of `HAEPG`, we recorded the time `HAEPG` spent for heap interaction modeling and exploit generation, as shown in Fig. 4. With the help of merging and pruning, `HAEPG` reduced the number of function paths significantly, and the total time for exploit generation of most programs is less than 400 s. The program with the most function paths without simplification is $airCraft$, which has 4284 function paths. Heap interaction modeling of this program without simplification took more than 10 h, while after removing the redundant function paths, it only left 6 of them and took 344 s for modeling. Besides, programs with complex algorithms, such as *note*3, which has a sophisticated bitwise algorithm, require a lot of analysis time due to complex constraints that need more time for solving.

Table 3. List of CTF pwn programs evaluated with `HAEPG`

Exp. Tech.	Name	CTF	Glibc ver.	Vul. type	Exp. prim.	Exp. gen.	Revery	M. Phish
Fastbin Attack	CaNaKMgF Remastered	ASIS CTF 2017	2.23	Double Free	AA	✔	✗	✗
	halconyheap	TJCTF 2019	2.23	Double Free	AA	✔	✗	✗
	stringer	RCTF 2018	2.23	Double Free	AA	✔	✗	✗
	secretgarden	Pwnable.tw	2.23	Double Free	AA	✔	✗	✗
	babyheap	Fireshell CTF 2019	2.24	UAF	AA	✔	✗	✗
	aircraft	RCTF 2017	2.24	UAF	AA	✗	✗	✗
Unsafe Unlink	stkof	HITCON 2014	2.23	Overflow	AW	✔	✗*	✗
	simple_note	Tokyo Westerns 2017	2.24	Off by One	AW	✔	✗*	✗
	fb	AliCTF 2016	2.24	Poison Null Byte	AW	✔	✗*	✗
	note3	ZCTF 2016	2.19	Overflow	AW	✔	✗*	✗
House of Force	gryffindor	InCTF 2017	2.23	Overflow	AX	✔	✗	✗
	bamboobox	NTU-CTF-2017	2.24	Overflow	AX	✔	✗	✗
Tcache Poisoning	three	BCTF 2018	2.27	UAF	AA	✔	✗	✗
	penpal world	RedpwnCTF 2019	2.27	UAF	AA	✔	✗	✗
	one	SECCON CTF 2019	2.27	Double Free	AA	✔	✗	✗
	girlfriend	StarCTF 2019	2.27	Double Free	AA	✔	✗	✗
	zero to hero	PicoCTF2019	2.27	Double Free	AA	✔	✗	✗
–	house_of_atum	BCTF 2018	2.27	UAF	–	✗	✗	✗
–	iz_heap_lv2	ISITDTU CTF 2019	2.27	Off by One	–	✗	✗	✗
–	schmaltz	InCTF 2019	2.28	Double Free	–	✗	✗	✗
–	children_tcache	HITCON 2018	2.27	Poison Null Byte	-	✗	✗	✗
-	Auir	CSAW CTF 2017	2.23	–	–	✗	✗	✗
–	Secret Note V2	HITCON 2018	2.23	–	–	✗	✗	✗
–	Car_Market	ASIS CTF 2016	2.23	–	–	✗	✗	✗

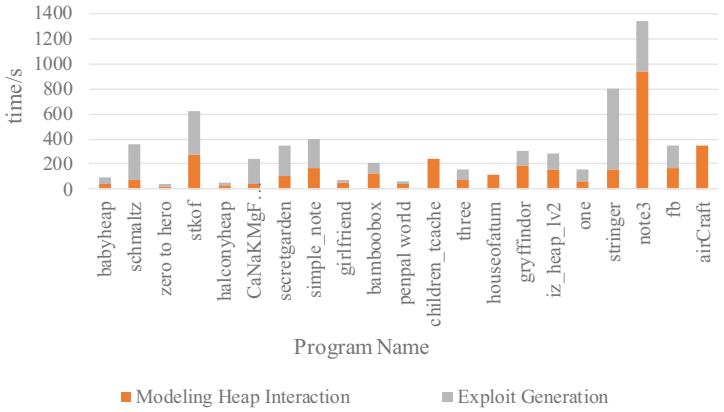

Fig. 4. Time intervals of modeling heap interaction and exploit generation

5.3 A Multi-hop Exploitation Case Study

We take *stkof* as an example and show how HAEPG automatically generates an exploit for a CTF challenge. The binary has three critical functions in *stkof*, *do_alloc*, *do_dealloc*, and *do_edit*, which are used for allocating, deallocating, and modifying heap chunks, respectively. An overflow in the *do_edit* function allows an attacker to write an arbitrary amount of data past the bounds of a chunk. Based on the vulnerability's capability, HAEPG used the template of unsafe unlink (as shown in Listing 1) for exploitation.

As shown in Fig. 5, HAEPG first generated *Attack Sequence 1* based on the unsafe unlink template. The template required the *vul_ptr* and *vic_ptr* to be adjacent when triggering the vulnerability. However, the glibc created *stdout_buffer* between the *vul_ptr* and *vic_ptr* unexpectedly when initializing io streams (HAEPG did not capture it because it only tracked heap interactions in *stkof* and ignored shared libraries). Thus HAEPG attempted to fix the heap layout by releasing the *stdout_buffer* first. Since *stdout_buffer* was generated by glibc, there is no function path that can release it. Hence, HAEPG tried to create a new heap layout. It constructed *Attack Sequence 2* by inserting a function path with an allocation primitive. HAEPG used the primitive to allocate a chunk *ph_ptr* in the same size as *vul_ptr*. The *ph_ptr* is allocated at *vul_ptr*'s original address and forced the *vul_ptr* and *vic_ptr* to be allocated at higher addresses, and they are adjacent to each other as desired. Now, the meta-data of *vic_ptr* can be corrupted through the overflowing from *vul_ptr*. Through the last deallocation, the heap allocator finally unlinked the fake chunk inside *vul_ptr* and caused an arbitrary write primitive. HAEPG corrupted *malloc_hook* with a one-gadget pointer via this primitive and hijacked the instruction pointer by inserting the function path of *do_alloc* into the *Attack Sequence* 2. When a shell was spawned, HAEPG generated the exploit input in S2E.

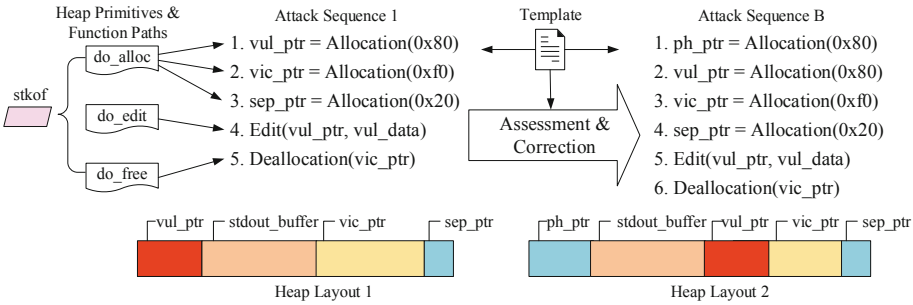

Fig. 5. Heap primitives to exploit *stkof*

5.4 Failed Cases

Failed on Exploit Generation: HAEPG corrupted a forward pointer of fastbin for *airCraft*. However, exploiting the AA primitive derived from fastbin attack requires a fake meta-data which is valid for size 0x88 in this case. HAEPG could not find such a primitive to store the illegal heap object, so it could not construct an exploitable state. To exploit the challenge, an attacker has to forge a fake chunk on the global data region in advance, which is beyond HAEPG's ability.

HAEPG failed on *house_of_atum* because the program only provides two heap objects for use at the same time, which are not sufficient for existing templates. The intended way is to confuse the tcache list and fastbin list.

Missing Capable Templates: HAEPG relies on templates for exploit generation, which means it can only handle the cases that are exploitable with existing templates, and can not use unknown exploitation techniques by itself. For instance, HAEPG can not handle *children_tcache*, *iz_heap_lv2*, and *schmaltz* because they are not exploitable with the provided templates. The intended solutions for *children_tcache*, *iz_heap_lv2* are to overlap heap objects by corrupting heap meta-data and triggering consolidation. As to *schmaltz*, the intended solution is to put a chunk in two different tcache lists and then corrupt the link pointer of it to get an AA primitive.

We downgraded the glibc version for them for further tests. HAEPG successfully generated exploit inputs for *iz_heap_lv2* and *schmaltz* with the templates of unsafe unlink and tcache poisoning. HAEPG could not solve *children_tcache* because the program only provides one chance to write for each object, which is not sufficient for provided templates.

Failed on Vulnerability Detection: Since HAEPG tracked heap interactions at the object level, it failed to detect memory corruptions in some cases. For instance, the challenge *Car_Market* has a buffer overflow inside objects, i.e., it will corrupt the adjacent data fields rather than adjacent objects. A more fine-grained method is needed to handle this case.

Failed on Program Analysis: HAEPG could not analyze sophisticated programs, such as *SecretNoteV2*, which has an AES encryption algorithm, and

Auir, which is an obfuscated program. As `HAEPG` relies on symbolic execution, these programs generated a large number of complex constraints and states, resulting in path/state explosion.

6 Related Work

Automatic Exploit Generation. Mechanical Phish [26] is a cyber reasoning system developed by the Shellphish team for DARPA's CGC [17]. It finds PoCs of vulnerabilities using Driller [29] and reproducing crashing states in Angr [27]. Then, it checks if input data corrupts write pointers or the instruction pointer at crashing points. If so, it will create shellcodes or rop-chains for exploitation. In the end, it solves data constraints and generates exploit inputs.

Revery [30] is an automatic exploit generation tool for heap-based vulnerabilities. It employes taint analysis and shadow memory to detect memory corruption in crashing inputs. Moreover, it searches for exploitable points using a layout-oriented fuzzing technique. In the end, Revery generates exploits by stitching the diverging paths and crashing paths together. As shown in Sect. 5, Revery failed on the evaluation set because the fuzzing technique that Revery used to explore exploitable points is not capable of multi-hop exploitation.

FUZE [31] is a novel framework to automate the process of kernel UAF exploitation. It analyzes and evaluates system calls which are valuable and useful for kernel UAF exploitation using kernel fuzzing and symbolic execution. Then, it leveraged dynamic tracing and SMT solver to guide the heap layout manipulation. The authors used 15 real-world vulnerabilities to demonstrate that FUZE could not only escalate kernel UAF exploitability but also diversify working exploits.

SLAKE [14] is a solution to exploit vulnerabilities in the Linux kernel. It uses a static-dynamic hybrid analysis to search for objects and syscalls which are useful for kernel exploitation. Then, SLAKE models the capability of vulnerability and matching the capability with corresponding objects. To exploit the vulnerability, SLAKE chooses the method of exploitation based on the vulnerability type and manipulates heap layouts by adjusting the free list in the slab.

Gollum [22] is the first approach to automatic exploit generation for heap overflows in interpreters. It employes a custom heap allocator SHAPESHIFTER to generate exploitable heap layouts and utilizes a genetic algorithm to find heap interaction sequences that can lead to the target heap layouts. If Gollum reaches the target heap layouts, it corrupts the function pointers of victim objects by triggering heap overflows. Like `HAEPG`, Gollum corrupts the instruction pointer with one-gadgets to generate exploits.

The solutions above which toward interpreter or kernel explore the heap state space with the knowledge of language grammars or kernel syscalls. However, there is no such standard input protocol for applications as each application parses inputs differently. Our method modeled heap interactions in the dimension of the program path, and the result showed the potency of it.

Besides, these solutions assume there is a sensitive pointer which could be overwritten by merely heap layout manipulation and triggering vulnerabilities. As we discussed before, the premise does not always hold for some vulnerabilities and applications, which makes exploitation harder. Our method could effectively solve this problem, and this is the main advantage of our method over others.

Moreover, instead of encoding existing exploitation techniques into `HAEPG`, we developed a templating language to abstract them. A user could write their templates without the need to know the internal details of `HAEPG`. The solutions mentioned above have no such interface.

Heap Exploitation Techniques Discovery. Heaphopper [19] is an automated approach to analyze the exploitability of heap implementations in the presence of memory corruption. It takes the binary library of heap implementation, a list of transactions (e.g., malloc and free), the maximum number of transactions that an attacker can perform, and a list of security properties as input. Heaphopper generates lists of transactions by enumerating permutations of the transactions and generate C files and compiled programs for them. Then, it executes these programs and tracks the memory states of them. If any program leads to states violating the security properties, Heaphopper will take the C file of it as output.

ARCHEAP [32] uses fuzzing rather than symbolic execution to discover new heap exploitation techniques. It generates test cases by mutating and synthesizing heap operations and attack capabilities, and checks whether the generated test cases can be potentially used to construct exploitation primitives, such as arbitrary write or overlapped chunks. As a result, ARCHEAP discovered five previously unknown exploitation primitives in ptmalloc and found several exploitation techniques against jemalloc, tcmalloc, and even custom heap allocators.

The solutions above focus on exploit techniques discovery rather than application, so they are usually used for heap allocator's security assessment. Our solution can generate exploits using known exploit techniques, but it can not make use of unknown exploit techniques.

7 Discussion

`HAEPG` is dedicated to automating the process of multi-hop exploitation for heap-based vulnerabilities. However, it has the following limitations:

- `HAEPG` could not analyze sophisticated programs or real-world programs. First, the static analysis which `HAEPG` used to extract CFG is not good at handling indirect calls/jumps. Second, symbolic execution's drawbacks make `HAEPG` only applicable to small programs. Third, a significant amount of real-world programs uses multi-threading or multi-processing, which brings additional challenges to the program analysis techniques used by `HAEPG`.
- `HAEPG` is fundamentally incomplete because it only searches for specific memory states based on existing templates rather than exploring the whole memory state space, which means `HAEPG` could not generate exploits with exploitation techniques where no template is given.

– HAEPG is implemented for ptmalloc and can not generate exploits for programs using other allocators for now. To adapt HAEPG to other heap allocators, we have to change the codes of parsing heap objects, the strategy of heap layout manipulation, and the codes of detecting and leveraging exploit primitives. We leave it as future work.

8 Conclusion

In this paper, we proposed an automatic exploit generation solution HAEPG for heap vulnerabilities, which uses hybrid techniques to build the heap interaction model and navigate the multi-hop exploitation. HAEPG could generate complex exploits that abuse heap allocator's internal functionalities and enhance the vulnerabilities' capability step by step, which previously could only be completed manually. We evaluated HAEPG with CTF challenges, and the result showed the effectiveness of HAEPG. In the end, we believe that HAEPG improves the state-of-the-art of automated exploit generation and provides useful building blocks for solving remaining challenges in the field.

References

1. Ctftime. https://ctftime.org/. Accessed 2020
2. Cve details. https://www.cvedetails.com/. Accessed 2019
3. Github. https://github.com/. Accessed 2020
4. The gnu c library (glibc). https://www.gnu.org/software/libc/. Accessed 2019
5. Heap exploitation - fastbin attack. https://0x00sec.org/t/heap-exploitation-fastbin-attack/3627. Accessed 2019
6. The malloc maleficarum glibc malloc exploitation techniques. https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt. Accessed 2020
7. Pwnable.tw. https://pwnable.tw/. Accessed 2020
8. tcache_poisoning.c. https://github.com/shellphish/how2heap/blob/master/glibc_2.26/tcache_poisoning.c. Accessed 2019
9. Unlink exploit. https://heap-exploitation.dhavalkapil.com/attacks/unlink_exploit.html. Accessed 2019
10. Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options (2017). https://github.com/google/honggfuzz. Accessed 2020
11. Cwe-193: Off-by-one error (2019). https://cwe.mitre.org/data/definitions/193.html. Accessed 2019
12. Avgerinos, T., Cha, S.K., Hao, B.L.T., Brumley, D.: AEG: Automatic exploit generation. In: 18th Network and Distributed System Security Symposium (2011)
13. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: 2012 IEEE Symposium on Security and Privacy. IEEE (2012)
14. Chen, Y., Xing, X.: Slake: facilitating slab manipulation for exploiting vulnerabilities in the Linux kernel. In: 2019 ACM SIGSAC Conference on Computer and Communications Security. ACM (2019)
15. Chipounov, V., Kuznetsov, V., Candea, G.: S2e: a platform for in-vivo multi-path analysis of software systems. In: ACM SIGARCH Computer Architecture News, vol. 39. ACM (2011)

16. cwe.mitre.org: Cwe-626: Null byte interaction error (poison null byte). https://cwe.mitre.org/data/definitions/626.html. Accessed 2019
17. DARPA: Cyber grand challenge. https://www.cybergrandchallenge.com/. Accessed 2019
18. david942j@217: [project] the one-gadget in glibc. https://david942j.blogspot.com/2017/02/project-one-gadget-in-glibc.html. Accessed 2019
19. Eckert, M., Bianchi, A., Wang, R., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Heaphopper: bringing bounded model checking to heap implementation security. In: 27th USENIX Security Symposium (USENIX Security 18) (2018)
20. Heelan, S.: Automatic generation of control flow hijacking exploits for software vulnerabilities. Ph.D. thesis, University of Oxford (2009)
21. Heelan, S., Melham, T., Kroening, D.: Automatic heap layout manipulation for exploitation. In: 27th USENIX Security Symposium (USENIX Security 18) (2018)
22. Heelan, S., Melham, T., Kroening, D.: Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In: 2019 ACM SIGSAC Conference on Computer and Communications Security. ACM (2019)
23. 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: 2012 IEEE Sixth International Conference on Software Security and Reliability. IEEE (2012)
24. PaX-Team: Pax address space layout randomization. https://pax.grsecurity.net/docs/aslr.txt. Accessed 2019
25. PaX-Team: Pax non-executable pages design & implementation. https://pax.grsecurity.net/docs/noexec.txt. Accessed 2019
26. Shoshitaishvili, Y., et al.: Mechanical phish: Resilient autonomous hacking. In: 2018 IEEE Symposium on Security and Privacy (2018)
27. Shoshitaishvili, Y., et al.: Sok: (state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy. IEEE (2016)
28. Sidhpurwala, H.: Hardening elf binaries using relocation read-only (relro). https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro. Accessed 2019
29. Stephens, N., et al.: Driller: augmenting fuzzing through selective symbolic execution. In: 23th Network and Distributed System Security Symposium, vol. 16 (2016)
30. Wang, Y., et al.: Revery: From proof-of-concept to exploitable. In: 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM (2018)
31. Wu, W., Chen, Y., Xu, J., Xing, X., Gong, X., Zou, W.: Fuze: towards facilitating exploit generation for kernel use-after-free vulnerabilities. In: 27th USENIX Security Symposium (USENIX Security 18) (2018)
32. Yun, I., Kapil, D., Kim, T.: Automatic techniques to systematically discover new heap exploitation primitives. arXiv preprint arXiv:1903.00503 (2019)
33. Zalewski, M.: American fuzzy lop (2017). https://lcamtuf.coredump.cx/afl/