Article

AAHEG: Automatic Advanced Heap Exploit Generation Based on Abstract Syntax Tree

Yu Wang [1], Yipeng Zhang [2,*] and Zhoujun Li [1]

[1] State Key Lab of Software Development Environment, Beihang University, Beijing 100191, China; wangyu777@buaa.edu.cn (Y.W.); lizj@buaa.edu.cn (Z.L.)
[2] School of Information Science and Technology, North China University of Technology, Beijing 100144, China
* Correspondence: zhangyipeng@ncut.edu.cn

Abstract: Automatic Exploit Generation (AEG) involves automatically discovering paths in a program that trigger vulnerabilities, thereby generating exploits. While there is considerable research on heap-related vulnerability detection, such as detecting Heap Overflow and Use After Free (UAF) vulnerabilities, among contemporary heap-automated exploit techniques, only certain automated exploit techniques can hijack program control flow to the shellcode. An important limitation of this approach is that it cannot effectively bypass Linux's protection mechanisms. To solve this problem, we introduced Automatic Advanced Heap Exploit Generation (AAHEG). It first applies symbolic execution to analyze heap-related primitives in files and then detects potential heap-related vulnerabilities without a source code. After identifying these vulnerabilities, AAHEG builds an exploit abstract syntax tree (AST) to identify one or more successful exploit strategies, such as fast bin attack and Safe-unlink. AAHEG then selects exploitable methods via an abstract syntax tree (AST) and performs final testing to produce the final exploit. AAHEG chose to generate advanced heap-related exploits because the exploits can bypass Linux protections. Basically, AAHEG can automatically detect heap-related vulnerabilities in binaries without source code, build an exploit AST, choose from a variety of advanced heap exploit methods, bypass all Linux protection mechanisms, and generate final file-form exploit based on pwntools which can pass local and remote testing. Experimental results show that AAHEG successfully completed vulnerability detection and exploit generation for 20 Capture The Flag (CTF) binary files, 11 of which have all protection mechanisms enabled.

Keywords: automatic exploit generation; heap-related vulnerability; fuzzing; symbolic execution; abstract syntax tree

check for updates

Citation: Wang, Y.; Zhang, Y.; Li, Z. AAHEG: Automatic Advanced Heap Exploit Generation Based on Abstract Syntax Tree. *Symmetry* **2023**, *15*, 2197. https://doi.org/10.3390/sym15122197

Academic Editor: Theodore E. Simos

Received: 1 November 2023
Revised: 30 November 2023
Accepted: 11 December 2023
Published: 14 December 2023

1. Introduction

Since the inception of the Cyber Grand Challenge hosted by DARPA in 2016 [1], Automatic Exploit Generation (AEG) has emerged as a focal point of research. Notable solutions, such as CRAX [2] and Mayhem [3], have been introduced to detect vulnerabilities in source code and binary files, and to autonomously generate vulnerabilities where feasible. The pivotal technologies employed in this domain are fuzzing and symbolic execution. While CRAX and Mayhem are well-established vulnerability exploitation systems, the majority of their attacks on the existence of vulnerabilities involve merely rudimentary concepts such as hijacking programs into shellcode. This utilization idea is ideal because it does not consider the protection mechanism opened by the binary file. For example, a simple protection mechanism, Non-Executable (NX) [4], can render this attack method ineffective.

At the same time, in the current research on vulnerability exploitation forms, most AEGs focus on stack overflow vulnerabilities and format string vulnerabilities. For example, the latest research, BofAEG [5] and LAEG [6], mainly target binary files with stack overflow vulnerability types. This is an asymmetrical phenomenon. As for heap-related AEG research, the more classic ones are Revery [7] and MAZE [8]. Their main problems are twofold. One is that the input needs to have a Proof of Concept (PoC), and the other is that some common heap advanced exploit forms such as *fast bin attack* or *tcache poisoning attack* are not applied. Although Revery can obtain an *unlink* state, it cannot successfully generate an exploit and utilize it. MAZE can perform partial heap utilization; however, with the support of PoC, it takes a relatively long time. At the same time, none of the above-mentioned AEGs can generate exploits in file form. The exploits generated in file form are very meaningful to assist experts in conducting in-depth research on the exploitation of this vulnerability.

In response to the current situation, we proposed Automatic Advanced Heap Exploit Generation (AAHEG). AAHEG focuses on mining heap-related vulnerability and solving the asymmetry in vulnerability mining and exploit generation. Its main contributions are as follows.

1. AAHEG applies symbolic execution technology to automatically collect information in binary files without the need for a PoC or source code. AAHEG automatically detects heap-related vulnerabilities (Heap Overflow, Off by One, Off by Null and Use After Free) existing in binary files.
2. By sorting out the current advanced exploitation methods of heap-related vulnerabilities targeting the Linux heap management mechanism glibc, and combining it with our daily attack experience, we extracted the vulnerability exploitation *abstract syntax tree (AST)* for each exploitation method. AAHEG will combine the protection mechanisms enabled, control flow graph (CFG) information in functions, and heap-related primitives in the binary file, and then select the appropriate exploit and complete the final exploit generation.
3. AAHEG can apply advanced heap utilization methods to bypass all protection mechanisms in Linux binaries and Linux systems, and use the Dynamic Payload Element (DPE) exploit generation strategy to bypass NX [4], PIE [9], Canary [10] and FULL RELRO [11].
4. AAHEG will eventually generate a file-form exploit based on the *pwntools* [12] tool to support subsequent expert research. At the same time, AAHEG will also use binary files and a remote docker for testing to verify the correctness of the generated exploit. Experimental results show that AAHEG can complete vulnerability detection and exploit generation for 20 Capture The Flag (CTF) binary files, 11 of which have the protection mechanism fully enabled.

The layout of this paper is as follows: Related work will be introduced in Section 2, an overview of AAHEG will be introduced in Section 3, the primitive extraction part of AAHEG will be introduced in Section 4, and Section 5 will introduce the vulnerability detection principle and work in AAHEG in detail. The vulnerability exploitation abstract syntax tree will be shown in Section 6. The details of the exploit generation will be described in Section 7. The experimental results will be shown in Section 8, and in Section 9, limitations and future work will be discussed. Conclusions will be presented in Section 10.

2. Related Work

2.1. Automatic Exploit Generation

Automated Exploit Generation (AEG) has always been a hot research topic. Traditional AEG generators mainly have the following methods. In order to solve this state of asymmetry in methods, where most AEGs only focus on methods from the Proof of Concept (PoC), we need to analyse the AEGs and then show the methods of how to generate the exploit from the binaries.

Starting from the Proof of Concept (PoC).Traditional AEGs require a vulnerability trigger path that has been found through fuzzing and other methods. Through this path, these AEGs determine how the vulnerability is triggered, and find the running state of the program when the vulnerability is triggered. Based on this runtime state, the basic form of the vulnerability and the method of exploit generation are determined. Research using this technical route includes the work of AEMB [13] and He's work [14]. AEMB finds PoC and converts the PoC into an exploit (EXP) that can bypass vulnerability mitigation measures. The main heap-related AEGs include Revery [7] and MAZE [8]. Revery will start from the PoC, and then use some directional fuzzing techniques to find the state that can control the *eip* register, so as to achieve the purpose of exploit generation. MAZE focuses on the heap layout problem; it uses the *Dig and Fill* algorithm to find a heap layout that can be exploited. Revery only supports the attack method of controlling the *eip* register, while MAZE can support *unlink* and *eip-hijack* attacks.

Instead of starting from the PoC, start directly from analyzing the binary. The main technologies used by this type of exploit generator are symbolic execution and taint analysis. There are many exploit generators that use symbolic execution technology, including CRAX [2], BofAEG [5] and AEG [15]. Symbolic execution technology can solve the problem of path constraint solving and can also cover more paths. After the exploit is generated, you can also use the Satisfiability Modulo Theories (SMT) solver to determine whether the *payload* can be written to the corresponding memory. Automatic exploit generators based on taint analysis mainly include the work of LAEG [6] and Huang [16]. Taint analysis mainly achieves the purpose of monitoring the data propagation process by monitoring taint propagation. LAEG analyzes and monitors the propagation of Canary values to bypass the Canary protection mechanism. Huang's work mainly uses taint analysis to monitor unsafe functions in input data and binary files.

Since the 2016 CGC, AEG has become a hot research topic. Even in a large global competition like CGC, the vulnerability exploitation environment is set up very ideally. For example, the ASLR protection mechanism of the system will not be turned on, and the NX protection mechanism will not be turned on in the binary file. This idealized environment preset makes vulnerability exploitation easier, and Automatic Exploit Generation against this preset environment is not powerful enough. For example, Revery only supports the *eip-hijack* attack method, which means that the prerequisite for this attack is to be able to implant shellcode in advance or to be able to find a backdoor in a binary file.

In daily attacks, all protection mechanisms can be bypassed by manually writing exploits, which means that automated exploit generation can also achieve this goal. Research on ways to bypass protection mechanisms is also a hot topic. For example, the exploits generated by HAEPG [17] can bypass both NX and Full RELRO protection mechanisms; AEGs such as BofAEG and CRAX based on stack overflow vulnerability can bypass the NX protection mechanism through *ret_to_libc*; AEMB can bypass the NX protection mechanism on Linux systems and the Data Execution Prevention (DEP) protection mechanism on Windows systems; LAEG uses the *taint analysis* method to track the location of the input and Canary value, and finally achieve the purpose of bypassing the Canary protection mechanism. But none of them can bypass all the protection mechanisms in Linux.

In heap-related AEG, there are two core research directions. One is the generation method of vulnerability exploitation, and the other is the construction of heap layout, also called *heap feng shui* [18]. Regarding the generation method of vulnerability exploits, most research focuses on *eip-hijack* and *unlink* attacks. MAZE can detect *Use After Free (UAF)* and *Heap Overflow* vulnerabilities and generate *eip-hijack* and *unlink* exploits. If it can only generate exploit based on *unlink* and it cannot bypass the PIE protection mechanism, ARCHEAP [19] can achieve the purpose of writing and constructing stacking blocks at any address by detecting whether a combination of a series of operations can be achieved.

Heap utilization research not only includes the selection of heap utilization methods but also the study of methods to achieve heap layout structure. In the research of heap layout, typical studies include the work of Zhang [20], the work of Heelan [21], the work of Gennissen [22], and the work of Li [23]. Zhang used distance-guided fuzzing to find specific heap layout structures, and their experimental results showed that they were able to generate 18 desired heap layout structures from 27 *Heap Overflow* vulnerabilities. Heelan proposed SHRIKE, a novel system that performs automatic heap layout operations on the PHP interpreter and can be used to construct control flow hijacking vulnerabilities.

Gennissen proposed *Hack the Heap*, a game that comes with an extensive tutorial to teach players how to play but does not require computer science knowledge. *Hack the Heap* also provides a toolchain that can generate heap vulnerability puzzles from unmodified real-world applications. *Hack the Heap* makes heap layout manipulation easier. Li proposed BAGUA. BAGUA first uses the heap operation dependency graph to accurately identify the primitives of heap layout operations and deeply analyzes their dependencies and capabilities. Based on this, it models the heap layout operation as an integer linear programming problem and solves the constraints to identify the sequence of primitives that achieves the desired heap layout.

AEG is also used in other architectures or other platforms. For example, Kang [24] proposed a framework for automatically catching exploits in JIT compilers, paying special attention to heap corruption vulnerabilities triggered by dynamic code (code generated by the JIT compiler at runtime). The purpose is to help assess the severity of a vulnerability and thereby assist in vulnerability classification. The attack target of EXGEN [25] is a smart contract, and EXGEN can generate a symbolic attack contract with partially sequential transactions and then symbolically execute the attack contract with the target to find and solve all constraints. teEther [26] also targets smart contracts, which allows the creation of exploits for contracts given only binary bytecode. Huang [27] proposed an end-to-end method that can create exploits based on crash input or existing exploits of various applications to generate exploits for some large applications. This is a huge breakthrough. AEG for the Linux kernel is [28], which introduced a new method, automatic vulnerability migration (AEM), to facilitate cross-version exploitability assessment for the Linux kernel. FUZE [29] proposed a new framework to facilitate the utilization process of kernel UAF.

Large Language Models (LLM) and deep learning are also current research hotspots. There are also many studies on the combination of LLM and Automatic Exploit Generation. For applications that utilize machine learning to automatically generate vulnerabilities, current work mainly focuses on the generation of shellcode or shellcode-based vulnerabilities. For the trained model to complete binary-level analysis work, the model needs to be trained to understand assembly code. Therefore, most of the work is based on translation, and indeed some work is based on pre-trained models in programming languages such as CodeBERT [30]. Translation-based work includes Neural Machine Translation (NMT) [31], which can efficiently generate assembly code for real shellcode starting from natural descriptions. DualSC [32] uses a shallow *Transformer* for model building and compares with six state-of-the-art baselines in terms of code generation and code summarization, which shows the competitiveness of DualSC. Additionally, there are other studies on using models to generate assembly code and Python code. For example, EVIL [33] can automatically generate exploit programs in assembly language and Python language based on natural language descriptions, and it also applies Neural Machine Translation (NMT) technology. ExploitGen [34] generates vulnerability exploit codes based on CodeBERT, which can effectively integrate template information into the semantics of the original natural language. ExploitGen can also generate Python code.

AAHEG mainly uses symbolic execution to extract primitives for functions in binary files. It determines the vulnerabilities in binary files through conditional judgments and directed symbolic execution in the primitives, and then generates a final exploit through vulnerability exploit abstract syntax trees (ASTs). Finally, with the help of constraint solving, AAHEG can generate the final exploit. Experimental results show that based on symbolic execution, AAHEG can complete vulnerability detection and exploit generation for 20 Capture The Flag (CTF) binary files, 11 of which have the protection mechanism fully enabled.

2.2. Heap-Related Vulnerability and Heap-Related Exploit

In glibc (ptmalloc) [35], whether it is a heap block in use or a heap block after release, the heap block structure used is rough, as shown in Figure 1.

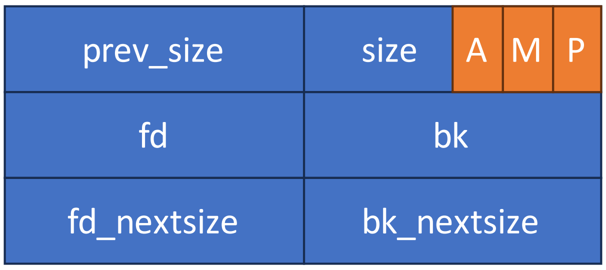

Figure 1. The heap structure in glibc.

The meaning of each field is as follows:

- prev_size. If the heap block before the physical address is in use, then this field has no meaning and can be used to store the data of the previous heap block; if the heap block before the physical address is not in use, then this field will be used to store the previous heap block's size.
- size. The size of the current heap block, of which the lower 3 bits are *A*, *M*, and *P* from high to low. *A* represents *NON_mainarena*, indicating whether the current heap block belongs to the main thread, where 0 means it belongs, and 1 means it does not belong; *M* means whether the current heap block is allocated by the mmap system call, 1 means it is allocated by the mmap system call [36] and 0 means not by the mmap system call(brk system call). *P* indicates whether the previous heap block is in use, 1 indicates that it is in use, and 0 indicates that it is not in use.
- fd/bk(forward/backward).If a heap block is in use, starting from *fd*, it is the data controlled by the user. If a heap block is not in use, then *fd/bk* will be used to save information related to the linked lists. *fd* represents the previous heap block (released before). *bk* represents the last heap block (released later). The order here is different between the fast bin chain and the tcache bin chain because the fast bin chain and the tcache bin chain use a single linked list structure (first in last out(FILO)). *fd/bk* is the core attack area for heap utilization. The attack targets of *unlink*, *fast bin attack*, and *unsorted bin attack* are all *fd/bk* area.
- fd_nextsize/bk_nextsize.These two fields will only be used in large bins. In heap exploitation, only *large bin attack* will attack these two fields.

There are four types of heap-related vulnerabilities. From the perspective of overflow, the most typical vulnerability is *Heap Overflow*. The root cause of *Heap Overflow* vulnerability is that the input data are not well controlled when writing data to the heap memory. For example, the requested length is 0x80 bytes, and the input length is greater than 0x80 bytes, which will cause a *Heap Overflow* vulnerability. Among the *Heap Overflow* vulnerabilities, there are two special ones. One is *Off by One*, and the other is *Off by Null*. The principles of these three vulnerabilities are shown in Figure 2. Another vulnerability is *Use After Free (UAF)*, where the heap block can still be indexed (read or written) after it is released.

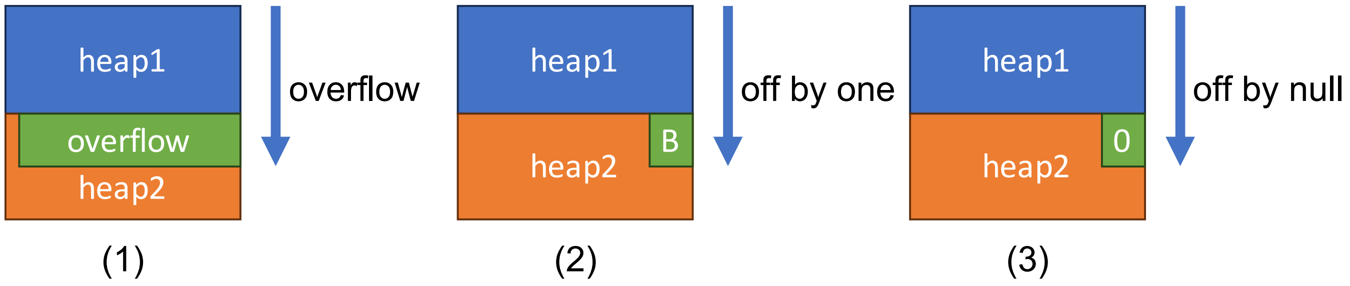

Figure 2. Heap-related vulnerability.

In this paper, heap-related utilization and its development history are introduced as follows.

- Unlink. *Unlink* is the oldest attack method. The first ones proposed were [37,38]. *Unlink* mainly uses heap-related vulnerabilities to overwrite linked-list-related data in the heap and then remove the heap blocks from the doubly linked list. When the link is broken, any address will be written. In 2004, a patch was proposed for this attack that detected link integrity in doubly linked lists, making the attack difficult to exploit.
- Safe-unlink [39]. The *Safe-unlink* method overwrites the *fd* and *bk* of a certain heap block P to *ptr – 0x18* and *ptr – 0x10* (ptr is the address of a pointer pointing to P) by laying out the heap block in advance, and then triggers the unlink mechanism of glibc. The ultimate goal of *Safe-unlink* is to write to an arbitrary address, and the condition is that the address of *ptr* needs to be known in advance, which means that the binary file needs to not have the PIE protection mechanism turned on.
- Fast bin attack [40]. The basic goal of *fast bin attack* is to modify the *fd* of a heap block that is already in the fast bin chain and then allocate two heap blocks of the same size. The basic goal of fast bin attack is to hijack `__malloc_hook` into one_gadget [41]. After that, we can hijack the control flow of the program.
- Tcache positioning [40]. Tcache is a mechanism added after Ubuntu 17.10 (glibc 2.26). It is mainly used to improve the efficiency of heap block allocation during program running. Tcache and fast bin are similar in data structure. *Tcache poisoning* refers to modifying the *fd* in the tcache chunk (heap block in the glibc heap manager) to the target. *Tcache poisoning* is an attack method of writing *fd* of chunk in the tcache bin, similar to *fast bin attack*.
- Unsorted bin attack [40]. The basic goal of the *unsorted bin attack* is to modify the *bk* of a heap block already in the unsorted bin chain to the target *address–0x10* (64 bit). After the modified unsorted bin is allocated, the target address will be written as *main_arena + 88* (a large value), unsorted bin attack is difficult to hijack control flow and needs to be coordinated with other exploitation methods.
- The House of Series. *The House of series* was first proposed by Phantasmagoria in *The Malloc Maleficarum* [42]. In *The Malloc Maleficarum*, five glibc attack methods including *The House of Force* were proposed. This naming method also affected subsequent naming methods for new exploits of glibc's heap manager, such as *The House of Orange* and *The House of Rabbit*.

In this paper, the test environment we used is Ubuntu 18.04 (glibc 2.27). Table 1 summarizes the feature, type, applicable versions, and protection mechanisms that can be bypassed for these exploits.

Table 1. Heap-related exploit methods in AAHEG.

Exploit Method	Feature	Type	Applicable Version	Protection Mechanism Bypass
Safe-unlink	Arbitrary address writing	Linked-list attack	Any version	NX, Canary, ASLR
fast bin attack	Control flow hijacking	Linked-list attack	Ubuntu version $\leq$ 20.04 (glibc version $\leq$ 2.31)	NX, PIE, Canary, ASLR
Tcahe poisoning	Arbitrary address writing/Control flow hijacking	Linked-list attack	Ubuntu version $\geq$ 18.04 (glibc version $\geq$ 2.27)	NX, PIE, Canary, ASLR
Unsorted bin attack	Write a large value at arbitrary address	Linked-list attack	Ubuntu version $\leq$ 18.04 (glibc version $\leq$ 2.27)	NX, PIE, Canary, ASLR

We did not put *The House of series* in Table 1. In our evaluation, we found that these utilization methods are not simple enough and can process too few binary files, so they are not worthy of being put into an AEG system. In AAHEG, it mainly applied *Linked-list attack* method shown in Table 1. The attack characteristics of *Linked-list attack* are simple and efficient. In many cases, we only need to modify the *fd/bk* of a chunk to achieve the final purpose of utilization.

3. Overview of AAHEG

3.1. Modules in AAHEG

AAHEG is divided into six modules in total.

1. Static analysis of binary files. The work to be carried out by static analysis of binary files is information collection. The work of information collection is to collect the protection mechanisms enabled in the binary file, such as whether PIE is enabled.
2. Primitive extraction. There are several aspects of primitive extraction including extraction of branch paths, conditions, and primitive grammar. Branch path extraction refers to extracting the conditions for reaching the `malloc`, `free`, and other functions. Simply put, primitive grammar is to determine the parameter range of functions' parameters such as heap block writing size, storage address, and other information in the malloc function.
3. Vulnerability analyzer. The vulnerability analyzer will solve the constraints of vulnerability based on the conditions existing in the primitive extracted before. For example, if you want to determine whether there is a *Heap Overflow* vulnerability, you need to determine whether the size written to a heap block after the `malloc` function exceeds its size `malloc` before. And it can be judged by the primitive grammar extracted before.
4. Exploit generator. The exploit generator will automatically search for an AST path that can successfully exploit the vulnerabilities found by the vulnerability analyzer. The ASTs are constructed based on experts' experience. The exploit generator collects protection mechanism information, generates the final exploit based on the abstract syntax tree, and verifies that the condition can be satisfied by the SMT solver.
5. Exploit verifier. The main work of the exploit verifier is to verify whether the vulnerability generated by the exploit generator can obtain the permissions of the target host. In AAHEG, the exploit verifier will automatically run a process or automatically start a remote docker to simulate the remote environment. If the exploit generator can successfully obtain the permissions of the target host, then the exploit verification is successful, otherwise, the exploit verification fails.
6. File-form exploit generator. File-form exploit generator will generate exploit files based on *pwntools*. First, a Python-based function is generated based on the primitive grammar, and then the code for the corresponding function is generated based on the AST branch selected by the exploit generator, and finally a complete exploit is generated. The file-form exploit generated is in Python language and it has a built-in gdb debugger to help experts do subsequent verification and analysis.

3.2. Attack Model

In AAHEG, users do not need any professional knowledge, they only need to provide the path of the binary file. AAHEG will receive the path of the binary file, automatically analyze the corresponding binary file, and finally generate the corresponding exploit.

3.3. Overview

The basic process of AAHEG is as follows.

(1) AAHEG obtains binary protection mechanism information through static analysis. (2) AAHEG obtains the primitives of the entire binary file and the grammar of the primitive through dynamic symbolic execution. (3) AAHEG analyzes the vulnerability existing in the binary file through primitive grammar. According to different vulnerability types, AAHEG generates corresponding exploits, such as leak-information exploit and hijack-control-flow exploit. The generation of the exploit is based on the grammar of primitives and the abstract vulnerability exploits AST. (4) AAHEG will verify the generated exploit. (5) AAHEG generates the exploit in file form after passing the verification. The exploit in file form can assist experts in subsequent verification and debugging.

For programs without vulnerabilities, AAHEG will not detect the vulnerability and will return normally. AAHEG will not generate wrong exploits because the exploit will be verified for exploitability at the end. AAHEG will start a process and then generate an exploit to verify whether it can obtain the *shell* of the target host. If it can be obtained, it means that the attack is successful, and if it cannot be obtained, it means that the attack failed. This kind of process is divided into two types; one starts a local process, and the other simulates a remote process (using docker simulation), represented by the *Local Exploit* and *Remote Exploit* in Figure 3.

The overview of AAHEG is shown in Figure 3.

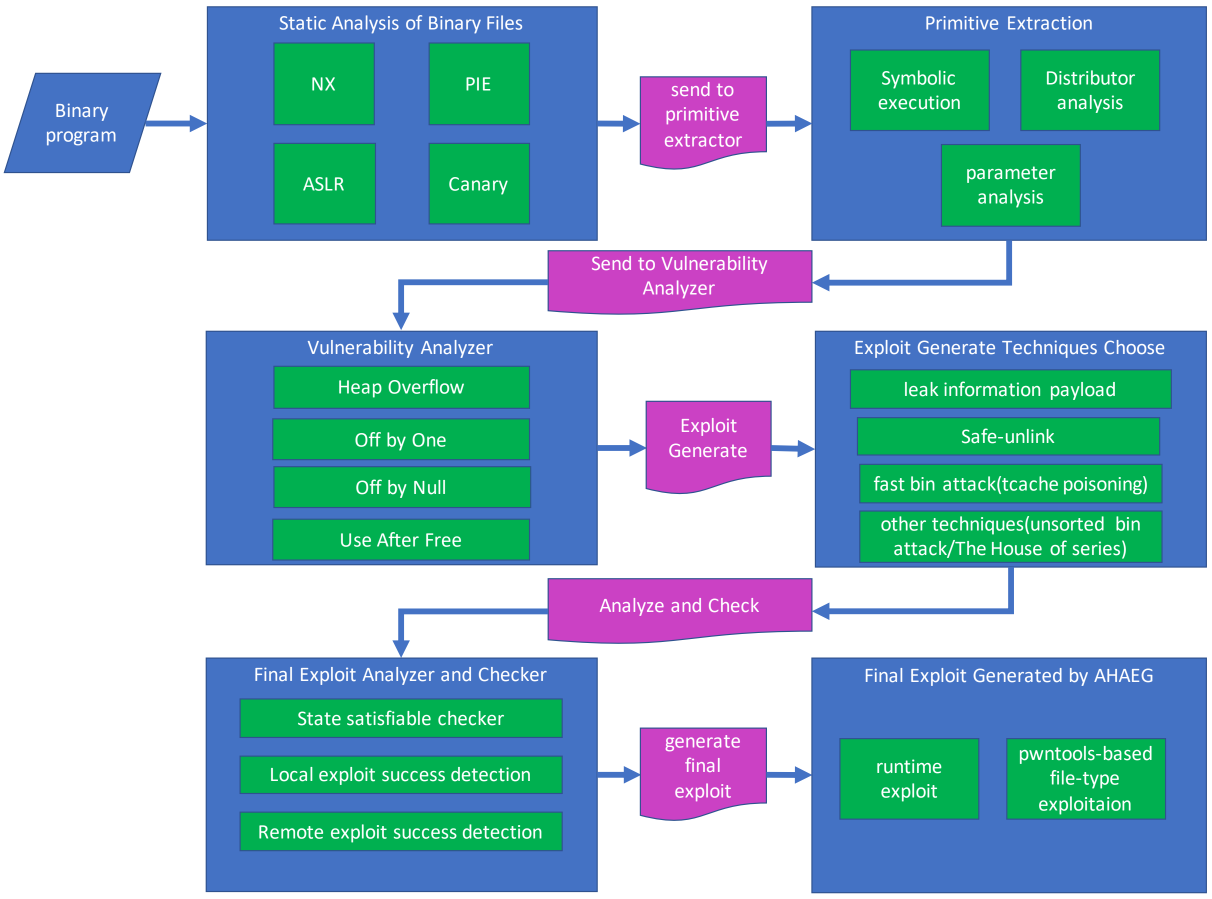
Binary program
Static Analysis of Binary Files
NX
PIE
ASLR
Canary
send to primitive extractor
Primitive Extraction
Symbolic execution
Distributor analysis
parameter analysis
Send to Vulnerability Analyzer
Vulnerability Analyzer
Heap Overflow
Off by One
Off by Null
Use After Free
Exploit Generate
Exploit Generate Techniques Choose
leak information payload
Safe-unlink
fast bin attack(tcache poisoning)
other techniques(unsorted bin attack/The House of series)
Analyze and Check
Final Exploit Analyzer and Checker
State satisfiable checker
Local exploit success detection
Remote exploit success detection
generate final exploit
Final Exploit Generated by AHAEG
runtime exploit
pwntools-based file-type exploitaion

Figure 3. Overview of AAHEG.

4. Primitive Extraction

Because AAHEG's attack model needs a binary file input and ultimately directly obtains a file-based exploit, the input binary file needs to be processed and analyzed. The first step in the processing here is to extract information such as protection mechanisms, and the subsequent work is primitive extraction.

Before introducing the basics of primitive extraction, we first determine the goals of AAHEG. AAHEG only targets event-loop-driven programs. The program is driven by the user input and enters different branches according to the user input. Different branches will have different effects, such as `malloc`, `free`, `show`, etc.

Figure 4 shows some types of binary files' functions that are mainly targeted by primitive extraction in AAHEG.

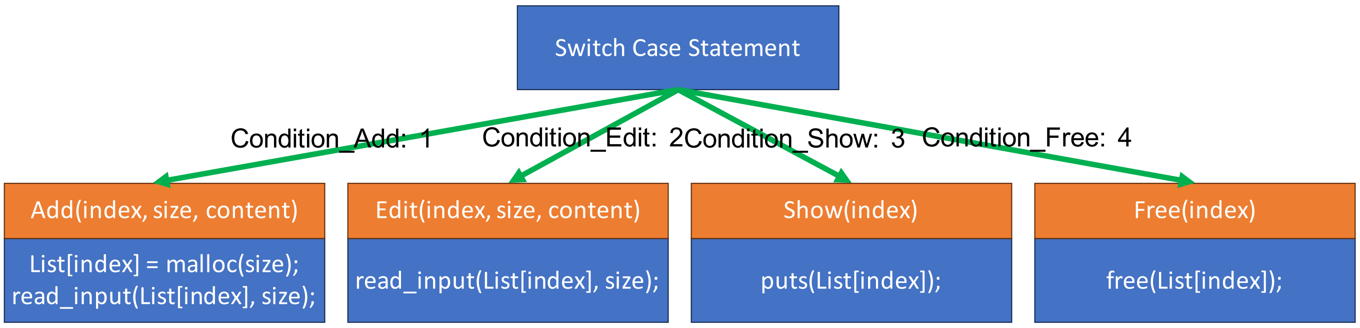

Figure 4. The function primitive to extract in binary files.

First, the program will select different branches based on the user's input. For example, to enter the `Add` branch, you need to enter 1, and to enter the `Edit` branch, you need to enter 2. The branches here are different in each binary and are dynamic. Then, there is the processing of the entire heap block. For example, Add mainly performs `malloc` and input data. The corresponding free is to release a heap block. `List` is the core storage area. `List` will save the addresses of all allocated heap blocks. `Show` function and `Free` function will treat List and index as direct parameters, and select the printed heap and released heap according to the specific value of the index function. Table 1 shows the characteristics of these functions.

Primitive extraction mainly consists of the following steps.

1. Constraint path extraction. AAHEG will first extract the constraint path, i.e., the path conditions from the entire binary file to each branch. The condition in Figure 4 has two main tasks. One is to perform the control flow graph (CFG) in the program. After analysis, the exit point of the branch statement that generates the switch case is extracted, which corresponds to the exit node of CFG. When extracting the exit node, you also need to prune the default branch of the switch statement or the error-reporting branch. The pruning method is to detect whether internal functions are called (external functions are puts or printf). Then, Angr [43], a symbolic execution tool, is used to solve the constraints of the corresponding branch path.
2. Function analysis. The second step of primitive extraction is to conduct an automated analysis of the function's functionality. The first step in automated analysis is to judge the `List` pointer. AAHEG will automatically search the `.bss` segment and other global segments to find the addresses indexed by the above functions (`Add`, `Edit`, etc.) and determine that this may be a `List` pointer. AAHEG uses two-way confirmation, i.e., after finding the possible `List` pointer, it continues to search for the possible `Add` function and finds the relevant address where `malloc` saves the return value in the `Add` function (determined based on the distance between the addresses, or based on the parameters) to determine whether it is an `Add` function and whether it is a `List` pointer. After the `List` pointer is determined, its content is determined based on the characteristics of other functions. For example, there is a call of the `free` function in `Free`, and the parameters of the `free` function are related to the `List` pointer; `Edit` does not call `malloc`, but the relevant address saved in the `List` is `Write`; the `Show` function will use some output functions to print the address related to the List.
3. Constraint paths correspond to functions. After extracting the corresponding constraint path and the corresponding function name, AAHEG will match these function names with paths and record them as the conditions of the function. Specifically, the information we want to extract here is the corresponding relationship in Figure 4. Entering 1 will enter the `Add` function, and entering 2 will enter the `Edit` function.
4. Key parameter information extraction. In AAHEG, the work of this step is to determine whether there is this parameter and the specific range of this parameter. For example, in the `Add` function, the specific value of `index` will be detected to determine its range. The detection here is based on the location where the last `malloc` return value is saved.

The next step is to determine the value range of the `malloc` parameter. The range here is determined directly by hooking the `malloc` function and using the SMT solver in Angr. After the function reaches the `malloc` function, its parameters will be probed to obtain their minimum and maximum values.

In terms of results, the value of the `Add` function can be fixed (corresponding to Fixed in Table 2) or a range (corresponding to Dynamic in Table 2). The final data that need to be extracted are the input content. The core data are the input length and input structure. The input length needs to be judged using Angr's *is_symbolic* attribute. *is_symbolic* represents whether data are variable, i.e., whether they are consistent with the input or variable data. If data have the *is_symbolic = True* attribute, it means that they may be related to the input. You can use the constraint solver in Angr to determine whether the application can be written. At the same time, you can also use *is_symbolic* to determine the length of the input. This size is very critical information, which is relevant to subsequent vulnerability detection. The input data here also pay attention to the last byte. The main purpose is to detect whether there are vulnerabilities such as *Off by One* or *Off by Null*. AAHEG's approach is to record the status of the memory application (malloc) and initialize it to uninitialized data. After data input, AAHEG will record the changed bytes, record their length, and record the state of the last byte. If they are uninitialized data, it proves that there is no vulnerability; if it is \x00, it should be an *Off by Null* vulnerability; if it is a symbolic state, it should be an *Off by One* vulnerability.

Table 2. Characteristics of functions.

Function Name	Index	Size	Content	Main Functions
Add	Optional	Dynamic/Fixed	Dynamic	Allocate a heap block
Edit	Dynamic	Optional	Dynamic	Modify the content of a heap block
Show	Dynamic	No	No	Print the content of a heap block
Free	Dynamic	No	No	Free a heap block

Optional: This parameter is optional, Dynamic: The value of this parameter is dynamic. Fixed: The value of this parameter is fixed. No: This function does not require this parameter.

The analysis process of other functions such as `Edit`, `Show`, and `Free` is similar to the analysis process of `Add` because the parameters in functions such as `Edit`, `Show`, and `Free` are similar to the `Add` function. Primitive extraction is a very important step in AAHEG. Because the attack model designed by AAHEG inputs a binary file instead of a PoC, it requires the support of reverse engineering tools. AAHEG uses radare2 [44] for reverse analysis, and radare2 can analyze the code, index, and symbol information in the binary file. radare2 also supports Python implementation, which is very suitable for AAHEG. Other AEGs, such as MAZE and other tools, require input from PoC, which is fundamentally different from AAHEG.

The work of primitive extraction is very important. First, the vulnerability type is determined through primitive extraction and key parameter range judgment. At the same time, when the exploit is generated, the key parameter range will also guide the program to select branches in the AST.

5. Vulnerability Detection

After completing the primitive extraction, AAHEG can basically understand the entire binary file. The first step in the follow-up work is to complete the detection of vulnerabilities. The overflow vulnerabilities are *Heap Overflow*, *Off by One* and *Off by Null*, and another type of heap-related vulnerability is UAF. This is the key concept of AAHEG because we solved the asymmetrical types of methods in vulnerability detection.

Heap Overflow. The detection of *Heap Overflow* vulnerabilities is relatively simple, because the input to the heap is directly in the `Add` and `Edit` functions, and the basic conditions are as Equation (1).

$$Add_{size} < input_{size} \vee ((Add_{size} < Edit_{size}) \wedge (Add_{index} == Edit_{index})) \tag{1}$$

In Equation (1), Add_{size} represents the allocated size, which is the parameter value of the malloc function; $input_{size}$ represents the length of the input; $Edit_{size}$ represents the length of the input in the edit function; Add_{index} represents the subscript index in the Add function, and $Edit_{index}$ represents the subscript index in the Edit function.

Because the range of parameters of the `malloc` function has been extracted during primitive extraction, AAHEG determines a representative value of the `malloc` function parameters based on the results of primitive extraction. After determining the representative value, if the input size exceeds the selected representative value, then it is determined that there is a *Heap Overflow* vulnerability, i.e., $Add_{size} < input_{size}$. If the size of `Add` is smaller than the size of Edit and the subscript of `Add` is equal to the subscript of `Edit`, it can also be determined that a *Heap Overflow* vulnerability exists, i.e., $Add_{size} < Edit_{size} \wedge Add_{index} == Edit_{index}$. If no vulnerability is found, AAHEG will also detect all value ranges and make judgments on all size values to ensure the existence of vulnerability.

Off by One/Off by Null. Because these two vulnerabilities are special types of *Heap Overflow* vulnerabilities. After AAHEG determines that the *Heap Overflow* vulnerability exists, it will also judge the overflow length of this vulnerability. Practically speaking, this judgment is determined by the length of symbolic data, which is the same as the judgment method of *Heap Overflow*. The judgment conditions for *Off by One* are as Equation (2).

$$Add_{size} + 1 == input_{size} \vee (Add_{size} + 1 == Edit_{size} \wedge Add_{index} == Edit_{index}) \tag{2}$$

The judgment condition for *Off by One* is the allocated length plus 1 is equal to the input length. Or the length in `Edit` is equal to the allocated length plus 1, and the subscripts of the `Add` function and the `Edit` function are consistent.

The judgment conditions for Off by Null are as Equation (3) or Equation (4).

$$Add_{size} + 1 == input_{size} \vee input_{size-1} == 0 \tag{3}$$

$$Add_{size} + 1 == Edit_{size} \vee Add_{index} == Edit_{index} \vee input_{size-1} == 0 \tag{4}$$

The judgment condition for *Off by Null* is that the allocated length plus 1 is equal to the input length, or the length in `Edit` is equal to the allocated length plus 1 and the subscripts of the `Add` function and the `Edit` function are equal. The difference between *Off by Null* and *Off by One* is that $input_{size-1} == 0$, i.e., the last character is fixed to 0 and cannot be any other character.

Use After Free. The strict definition of UAF is that a heap block can still be indexed after it is released. The common method of exploitation is to directly modify the content or release it again (Double Free). To address this vulnerability, AAHEG's judgment condition is Equation (5).

$$Free_{index} == Show_{index} \vee Free_{index} == Edit_{index} \tag{5}$$

That is, the subscript of `Free` is equal to the subscript of the `Show` function or the subscript of `Edit`. Practically speaking, this detection method is indirect. The actual detection conditions are:

$$Free_{addr} == Show_{addr} \vee Free_{addr} == Edit_{addr} \tag{6}$$

The addr here represents the address of the heap block, i.e., after a heap block is released, it can also be indexed by the `Show` function or the `Edit` function. If there are no problems with the primitive extraction steps, then the meaning expressed by the two formulas is the same, because the relationship between the index and addr is a one-to-one correspondence.

This section introduces the vulnerability detection method used in AAHEG. The detection idea is based on the correctness of primitive extraction.

6. Exploit Abstract Syntax Tree

After AAHEG completes vulnerability detection, it is necessary to pre-evaluate and select the exploitation method. In AAHEG, whether a vulnerability can be exploited is determined through an AST. This AST is based on our team's experience in heap-related exploitation. When we exploit a certain binary file, we first obtain the primitives of each function through reverse analysis. After extracting the primitives, we use some restrictions in the primitives (such as limiting the allocation size, input length, etc.) to choose the appropriate exploitation method. These exploitation methods are divided into the following parts: exploitation of leaking the libc, exploitation of hijacking hooks, and exploitation of triggering hooks. It is worth mentioning that because the higher version of glibc and the test environment version used by AAHEG are different, AAHEG uses glibc 2.27. The AST version introduced in this paper is glibc 2.27. For higher versions, just following this principle of syntax tree generation is enough to write the corresponding AST.

For example, when using glibc 2.29, the following code shows the protection mechanism added.

```
    p = chunk_at_offset(p, -((long) prevsize));
if (__glibc_unlikely (chunksize(p) != prevsize))
malloc_printerr ("corrupted size vs. prev_size while consolidating");
```

Glibc will check whether the size of the heap block with the first physical address is the same as the *prevsize* field. As a result, the syntax tree for *Off by Null* may change in glibc version 2.29. Specifically, the heap address needs to be leaked in advance during utilization in glibc version 2.29.

In AAHEG, we define three operation types:

- Sequence type (S): Sequence type, which means that these two operations are executed sequentially, i.e., executed one after another, and the left subtree precedes the right subtree.
- Multiplication type (M): Multiplication type, representing the number of times the right subtree repeats the left subtree operation.
- Conditions (C): Condition type, which means that when performing the operation of the left subtree, the conditions of the right subtree must be met.

In the AST, the four function types will be simplified, and the four functions `Add`, `Edit`, `Free`, and `Show` will be simplified to A, E, F, and S, respectively. As a parameter of the Add function, si(i = 0, 1, 2...) represents the size of the allocation, corresponding to the allocation order; i(i = 0, 1, 2...) represents the subscript. pi(i = 0, 1, 2...) represents the order of input payload. For example, in A(s0), s0 represents the size of the first allocation, and F(0) represents the release of the 0th allocated heap block.

Leak libc Exploit. We directly show the abstract syntax tree through UAF vulnerability, as shown in Figure 5.

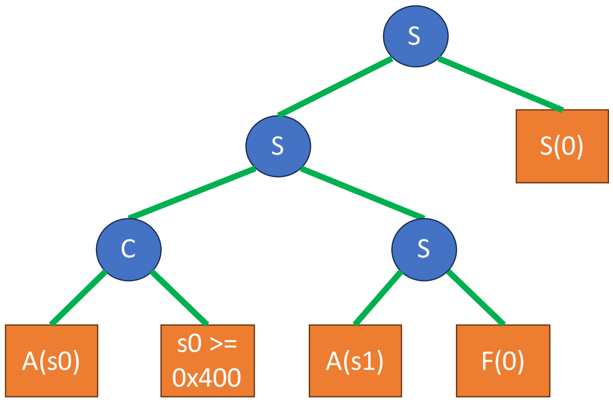

Figure 5. The Leak Libc AST through UAF vulnerability.

The sequence of operations represented by UAF AST is 1. A(s0) & s0 >= 0x400 2. A(s1) 3. F(0) 4. S(0). Because of the existence of UAF vulnerability, after freeing the heap block corresponding to the subscript 0, it can still be indexed. The condition of s0 >= 0x400 here is to bypass tcache (because the maximum size of tcache bin is 0x400), so that the released heap blocks will be put in the unsorted bin, and the libc-related address can be obtained by directly printing the content of heap block 0.

At the same time, the AST of the Leak Libc through *Heap Overflow* vulnerability is given, as shown in Figure 6.

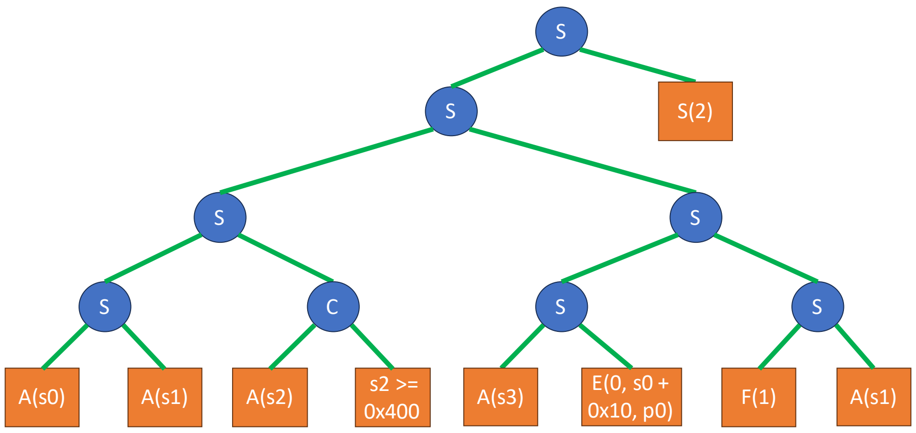

Figure 6. The Leak Libc AST through Heap Overflow vulnerability.

The sequence of operations represented by the *Heap Overflow* AST is 1. A(s0) 2. A(s1) 3. A(s2) & s2 >= 0x400 4. A(s3) 5. E(0, s0 + 0x10, p1) 6. F(1) 7. A(s1) 8. S(2). Because of the *Heap Overflow* vulnerability, we can directly modify the size of heap block 1 through overflow heap block 0. p1 is "\00" * s0 + p64(0) + p64(s0 + 0x10), where p64 is function pack 64, which represents re-unpacking the value into an 8-byte string, which corresponds to the endian order of the machine (big endian or little endian). The condition of s0 >= 0x400 here is also to bypass the tcache mechanism and allow the released heap blocks to be put in an unsorted bin. The function of F(1) and A(s1) is to release the forged heap blocks first, and then A(s1). Then, the unsorted bin will be at the same location as the heap block 2, and S(2) can leak the libc-related address.

Exploitation of hijack hooks. In glibc version 2.27, the main attack target of AAHEG is `__free_hook`. If `__free_hook` can be hijacked as the `system` function, then finally freeing a heap block with the content of "/bin/sh" is equivalent to executing `system("/bin/sh")`.

The AST is given directly here, as shown in Figure 7.

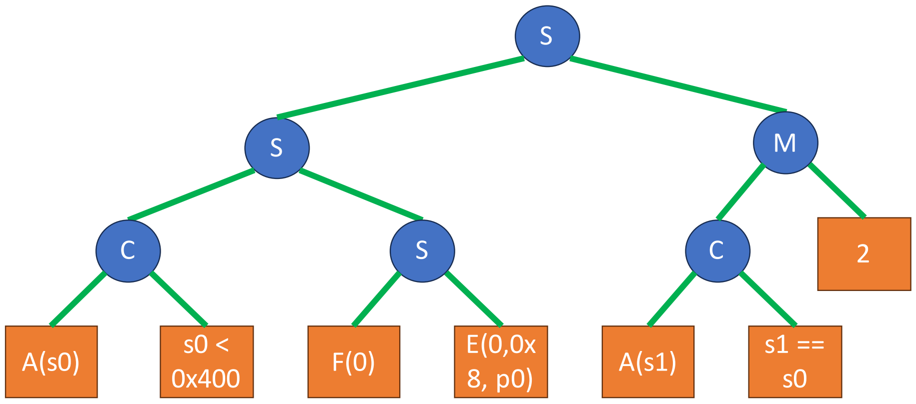

Figure 7. The Hijack Hooks AST through UAF vulnerability.

The sequence of operations represented by the UAF AST is 1. A(s0) & s0 < 0x400 2. F(0) 3. E(0, 0x8, p0) 4. (A(s1) & s1 == s0) * 2. Because of the existence of UAF vulnerability, after allocating and releasing the heap block corresponding to the free subscript 0, the heap block 0 will enter the tcache chain, and then allocate heap blocks of the same size twice in a row. p0 here is the payload, which is `p64(__free_hook)`, i.e., the *fd* location of tcache is overwritten to the address of `__free_hook`. The final allocated heap block will hijack `__free_hook` as the address of the system function.

The AST of *Heap Overflow* is shown in Figure 8.

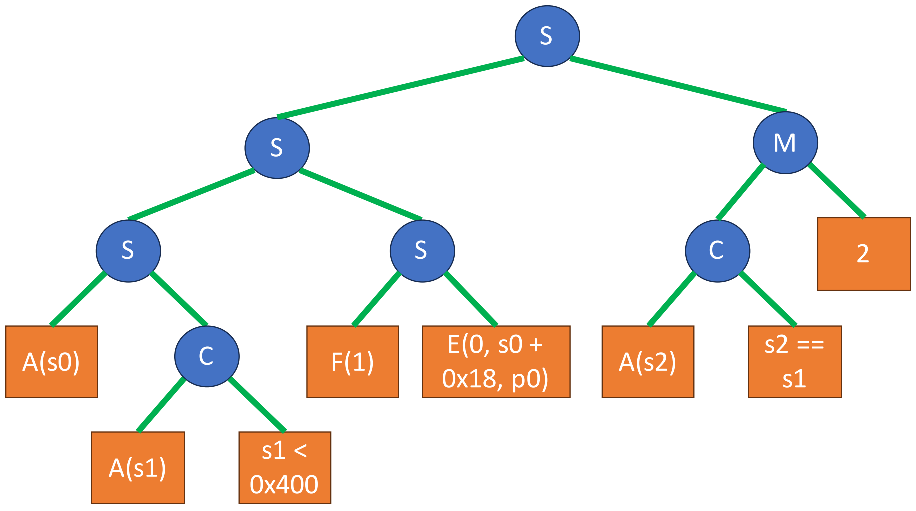

Figure 8. The Hijack Hooks AST through Heap Overflow vulnerability.

The sequence of operations represented by the *Heap Overflow* AST is 1. A(s0) 2. A(s1) & s1 < 0x400 3. F(1) 4. E(0, s0 + 0x10, p0) 5. (A(s2) & s2 == s1) * 2. Because of the existence of a *Heap Overflow* vulnerability, after allocating and releasing the heap block corresponding to the free subscript 0, the heap block 0 will enter the tcache chain, and then allocate heap blocks of the same size twice in a row. p0 here is the payload, which is `p64(__free_hook)`, i.e., the *fd* location of tcache is overwritten to the address of `__free_hook`. The final allocated heap block will hijack `__free_hook` as the address of the system function.

For the two vulnerabilities *Off by One*/*Off by Null*, the corresponding exploitation methods will be different. Here, we directly give the utilization AST of the `Off by Null` vulnerability, which is the utilization AST of hijacking `__free_hook`. Because *Off by Null* can be used, *Off by One* can also be used, because the bytes overflowed by *Off by One* are arbitrary bytes. The AST for using `Off by Null` is given in Figure 9.

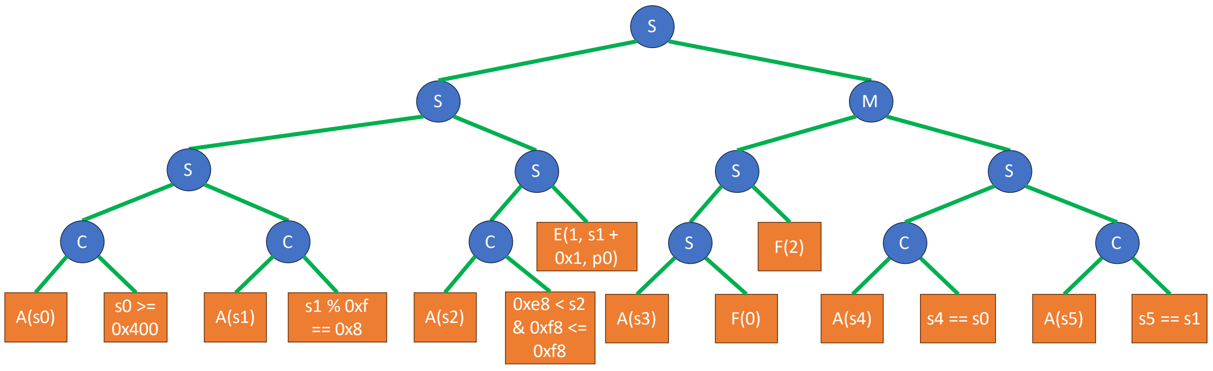

Figure 9. The Hijack Hooks AST through Off by Null vulnerability.

The operation sequence represented by the AST is 1. A(s0) & s0 >= 0x400 2. A(s1) & s1 % 0xf == 0x8 3. A(s2) & 0xE8 < (s2 & 0xf8) <= 0xf8 4. E(1, s1 + 0x1, p0) 5. A(s3) 6. F(0) 7. F(2) 8. A(s4) & s4 == s0 9. A(s5) & s5 == s1. The first is to lay out the heap, first allocate four heap blocks, and then modify the prev_size and size fields of subsequent heap blocks through *Off By Null*. F(0) and then F(2) trigger the merger of the two heap blocks, and then allocate one Heap block can make heap 1 and heap 5 at the same location.

The function of the AST is to give AAHEG a formalized way. AAHEG can pre-verify the exploit generation through the exploit AST and guide the generation of exploits through the exploit AST. The S operation can guide AAHEG to the next detection algorithm. The C operation can guide AAHEG to constraint solver to some conditions. The M operation is similar to the S operation and is used to guide AAHEG's utilization detection process.

It is worth mentioning that in this paper, only some typical vulnerability exploitation ASTs are listed. In AAHEG, there are dozens of actual vulnerability generation ASTs, because some details will be different, such as UAF leaks. When libc-related address data are needed, you can choose to continuously free more than seven heap blocks with sizes smaller than 0x400 and larger than 0x80, and then leak the information. The corresponding AST is shown in Figure 10.

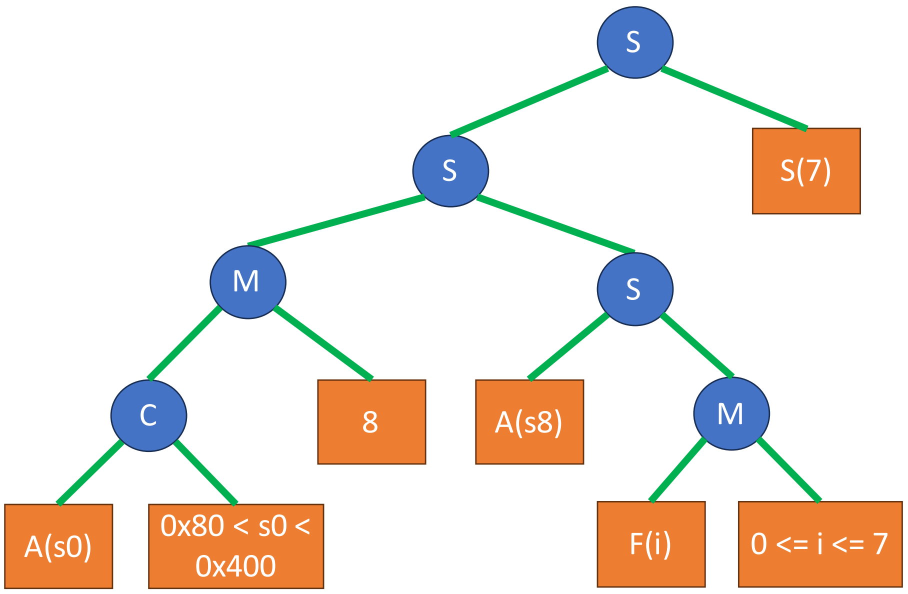

Figure 10. The Leak Libc AST through UAF vulnerability with bypassing tcache.

First, allocate nine heap blocks, as long as the size meets 0x80 < size < 0x400, then release the first eight heap blocks, so that the first seven heap blocks will fill the tcache chain, and the last heap block will enter the unsorted bin, S(7) will leak the libc-related address information when printing this heap block.

In short, the exploit AST is used to abstract heap-related exploits. The advantage of this abstraction is that it can help AAHEG select the exploit method and pre-verify whether the exploit can be generated, and finally complete the generation of the exploit.

7. Exploit Generation

Exploit generation is the final work of AAHEG. Its core work is to obtain the extracted primitives after obtaining the symbol information and path information, analyze the vulnerabilities detected in the binary file, choose an AST based on the vulnerability exploit, and finally generate a feasible exploit. Finally, AAHEG will generate the exploit in the form of file to give the experts more information about vulnerabilities. By generating the file-form exploit, experts can improve their ability to defend against vulnerabilities, because they have exploit files to debug to see the detailed part of vulnerabilities. By this file-form exploit, AAHEG helps the defenders, thus ensuring the symmetry of network security.

Dynamic Payload Element (DPE). In AAHEG, in order to bypass the two protection mechanisms, PIE and ASLR. We adopt the Dynamic Payload Element strategy. DPE will classify and mark all content in the payload. The purpose of this processing is to generate a dynamic payload, and then it is only instantiated when it is actually generated. Because in actual attacks, the first thing to do in an exploit is to leak the address to bypass the PIE and ASLR mechanisms, and then exploit it. For example, in heap utilization, the first thing to do is to leak the base address of the heap allocation and the base address of the libc library. Only then can the actual address of the content that needs to be hijacked be known, such as the address of `__free_hook` and the address of system function.

There are two basic operating conditions for DPE:

- Mark the elements in the generated payload and mark the address information that needs to be known in advance when instantiating it.
- The preamble payload is responsible for leaking information that subsequent payloads need to know in advance. If the required address information is not known when the payload is instantiated, the payload will fail to instantiate and need to be regenerated.

DPE is consistent with the actual attack model, which leaks the address first and then instantiates it. In DPE, each payload contains three elements—value, addr, and type—where value represents the specific value, addr represents the required address information, and type represents its numerical type.

For example, in the *Safe-unlink* attack, AAHEG obtained the pointer address *ptr* pointing to the heap block through binary static analysis. According to the basic Safe-unlink attack syntax, the following payload was obtained:

(0, None, int64), (0x21, None, int64), (ptr − 0x18, “elf_base”, int64), (ptr − 0x10, “elf_base”, int64), (0x20, None, int64), (0x30, None, int64)

The 0, 0x21, and other data here represent its numerical information; none means that it does not need to know the address information, “elf_base” means that this payload needs to know the base address of the binary file loading in advance; int64 represents 64-bit integer type data.

Other address types include none, no address type required; “elf_base” binary file loading base address; “Canary” needs to know the Canary value in advance; “libc_base” needs to know the libc loading base address in advance.

Specifically, for example, in an attack like *Safe-unlink*, if the binary file does not have the PIE protection mechanism turned on, then AAHEG will convert all the data related to “elf_base” in the generated payload into the none type, such as the payload above. will be converted to:

(0, None, int64), (0x21, None, int64), (ptr – 0x18, None, int64), (ptr – 0x10, None, int64), (0x20, None, int64), (0x30, None, int64)

The payload of the hijacking `__free_hook` generated in AAHEG is as follows:

(libc.symbols[“__free_hook”], “libc_base”, int64) libc.symbols["__free_hook"] represents the address of the `__free_hook` symbol obtained from the *libc.so* link library file, and "libc_base" represents the loading base address of the *libc.so* link library that needs to be known in advance.

DPE is used to assist in exploit generation and can help AAHEG complete the bypass of some protection mechanisms that exist on Linux. With DPE, combined with the work of the pre-order payload, AAHEG is able to bypass all protection mechanisms on Linux.

The final exploit generation flow chart is given in Figure 11.

After having the exploit generation syntax tree and DPE, AAHEG has the ability to generate exploits. Finally, AAHEG will start a process to verify whether the entire payload is correct (Check Local), and start a docker to verify whether the entire payload is correct (Check Remote).

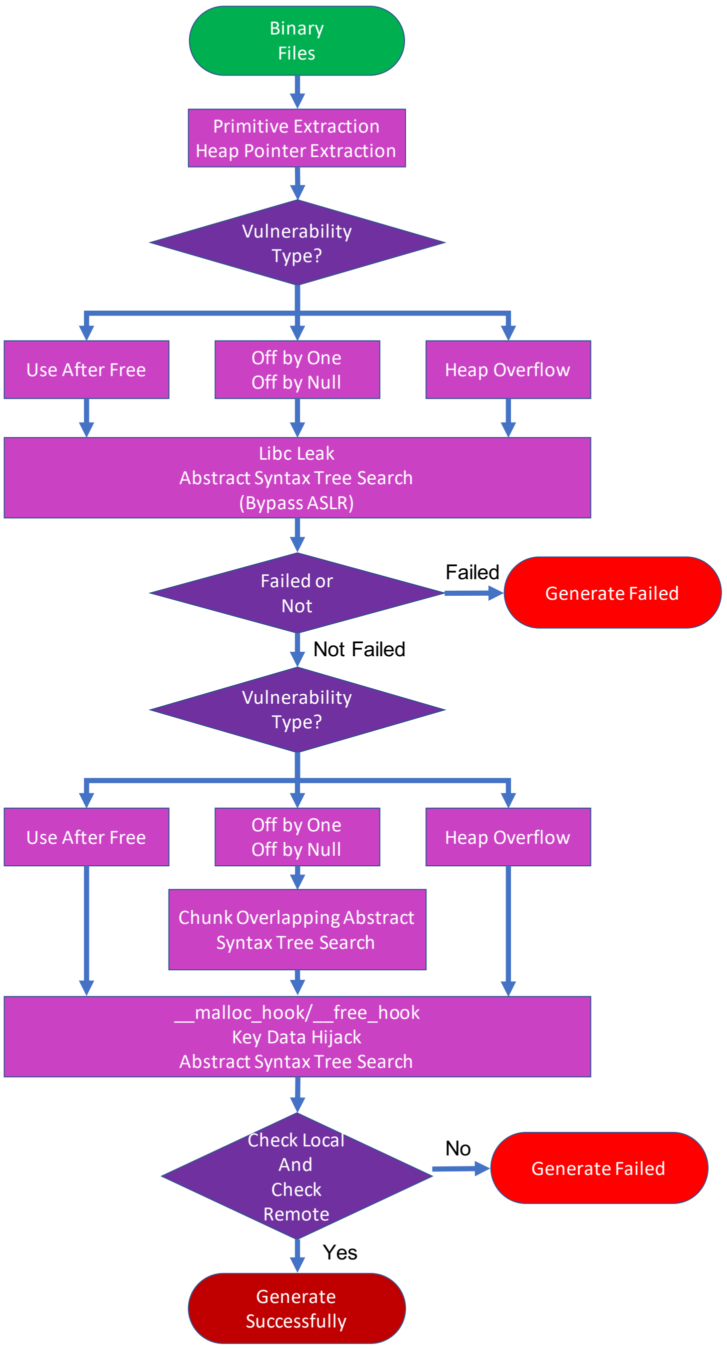
Binary Files
Primitive Extraction Heap Pointer Extraction
Vulnerability Type?
Use After Free
Off by One Off by Null
Heap Overflow
Libc Leak Abstract Syntax Tree Search (Bypass ASLR)
Failed or Not
Failed
Generate Failed
Not Failed
Vulnerability Type?
Use After Free
Off by One Off by Null
Heap Overflow
Chunk Overlapping Abstract Syntax Tree Search
__malloc_hook/__free_hook Key Data Hijack Abstract Syntax Tree Search
Check Local And Check Remote
No
Generate Failed
Yes
Generate Successfully

Figure 11. The flow chart of AAHEG.

8. Evaluation

8.1. Experimental Setup

Based on vulnerability detection and detection, we implemented these ideas in AAHEG. A total of 20,000 lines of code were written to complete these methods, of which about 3000 lines of code were the dynamic execution engine we completed based on Angr, about 4000 lines of code were the primitive extractor based on Angr, and the remaining 13,000 lines of code implemented AAHEG. These codes also implement database saving results and simulation of the docker environment.

We used Angr version 9.1.11752 for symbolic execution, pwntools 4.7.0 for binary analysis and exploit generation, and radare2 5.8.3 to assist with reverse engineering. The experimental environment uses Ubuntu 18.04 64-bit operating system and Intel Core i7 chip, 8 GB memory, and Linux kernel version 5.4.0.

8.2. CTF Benchmark Evaluation

Then we selected 20 CTF problems. Most of these CTF topics are from ctftime [45]. CTF problems can be considered as simplified versions of real programs for more concise and demonstrate the principle of vulnerability. The difference is that the CTF problems are used to test the relevant abilities of the players in the competition, so the vulnerabilities can be exploited, but the real-world programs are more complex, and even if there are vulnerabilities, they may not necessarily be exploitable. Table 3 shows our experimental results.

Table 3. Evaluation AAHEG by 20 CTF binary files.

Binary Name	Vuln Type	N	C	P	R	Advanced Exploit Technique	Exploit	T(s)
2019_5thspace_final_pwn1	UAF	✓	✓	✓	F	Tcache poisoning	L + R	32
2020_diaoyucheng_very_easy	UAF	✓	✓	✓	F	Tcache poisoning	L + R	50
2020_tiesan_fake	UAF	✓	✓	×	P	Safe-unlink	L + R	25
2020_wangding_magic	UAF	✓	✓	×	P	Safe-unlink	L + R	37
2023_longjian_14	UAF	✓	✓	✓	F	Tcache poisoning	L + R	41
2023_longjian_8	UAF	✓	✓	×	P	Safe-unlink	L + R	47
2017_RCTF_RNote	Off by One	✓	×	×	P	Safe-unlink	L + R	57
2018_0ctfquals_babyheap	Off by One	✓	✓	✓	F	Fast bin attack	L + R	78
2019_roarctf_easy_pwn	Off by One	✓	✓	✓	F	Tcache poisoning	L + R	63
2023_longjian_16	Off by Null	✓	✓	✓	F	Tcache poisoning	L + R	23
2018_LCTF_easy_heap	Off by Null	✓	✓	✓	F	Tcache poisoning	L + R	73
2018_qctf_babyheap	Off by Null	✓	✓	✓	F	Fast bin attack	L + R	76
2018_rctf_babyheap	Off by Null	✓	✓	✓	F	Fast bin attack	L + R	88
2019_5thspace_pwn12	Off by Null	✓	✓	×	F	Safe-unlink	L + R	42
2019_5thspace_pwn14	Off by Null	✓	✓	×	P	Safe-unlink	L + R	31
2019_swamp_dream_heaps	Off by Null	✓	✓	×	P	Safe-unlink	L + R	54
2019_CISCN_pwn8	Heap Overflow	✓	✓	×	P	Safe-unlink	L + R	24
2019_qiangwang_AP	Heap Overflow	✓	✓	✓	F	Tcache poisoning	L + R	51
2020_diaoyucheng_unknown	Heap Overflow	✓	✓	✓	F	Tcache poisoning	L + R	34
2021_longjing_hellocat	Heap Overflow	✓	✓	×	P	Safe-unlink	L + R	49

for table header: N: NX, C: Canary, P: PIE, R: RELRO, T: the time consumed by AAHEG, L means local check succeeded, R means remote check succeeded(attack remote docker).

As can be seen from Table 3, AAHEG can detect 4 types of vulnerabilities—UAF, *Heap Overflow*, *Off by One* and *Off by Null*. This detection result demonstrates the completeness of AAHEG's vulnerability detection and also verifies AAHEG's ability to detect four types of heap-related vulnerabilities. At the same time, it can be seen from Table 3 that AAHEG chooses different exploit generation methods based on these four vulnerabilities. For example, through UAF vulnerabilities, AAHEG will choose different exploit methods based on the protection mechanism enabled by the actual binary file. When PIE is enabled, the *Safe-unlink* exploit is powerless. When PIE is not enabled, AAHEG will choose to generate the *Safe-unlink* exploit.

Overall, AAHEG's exploit generation time does not exceed 2 min, which is a huge improvement. At the same time, 11 of the 20 CTF binary files have all protection mechanisms enabled, which shows that AAHEG can bypass all protection mechanisms in Linux systems.

9. Discussion

AAHEG has the ability to bypass the protection mechanisms in Linux and it can successfully generate the exploit of heap-related vulnerabilities. However, AAHEG still has some limitations and issues to overcome, so we discuss these limitations and future work.

9.1. Limitations

Heap Layout Problem. AAHEG does not consider heap layout problems before exploit generation. We suggest an ideal environment where the primitive extraction is strict. The noise of heap allocating can be eliminated by the AST, because through the AST, AAHEG can finally find a path that eliminates the noise.

Limitation of symbolic execution. AAHEG applies symbolic execution to extract the primitives of binary files. It means AAHEG is sensitive to function complexity when the path condition of primitives is, respectively, complex. Fortunately, it does not matter when the program is not too large such as a CTF problem. But when we want to apply AAHEG to some real-world problem, we need to optimize symbolic execution.

9.2. Future Work

Handle real-world problem. We will apply fuzzing technology to extract the path of heap-related primitives. For example, when extracting a primitive of a real-world problem, we can use fuzzing to find the path of a function call (`malloc`, `free`). Then, we use symbolic execution to extract the primitives of this path. Finally, AAHEG can find the primitive and use the primitive to generate an exploit.

Solve higher versions of glibc. When the glibc version is higher than 2.33 (or equal to 2.33), glibc will adopt a `safe-linking` mechanism to protect the *fd* of the heap block. Its basic logic is to *XOR* the *fd* with the address of the current heap block, which increases the difficulty of automated heap utilization. We need to add leakage of heap block addresses in the exploit AST in advance. The difficulty of this in actual application will not increase too much; it is just a matter of changing the general exploit AST.

9.3. Related Work

Heap-related vulnerabilities mining. Some research focuses on improving the efficiency of finding heap-related vulnerabilities. Tu [46] designed and implemented an enhanced symbolic execution engine named HEAPX to facilitate the automatic detection and exploitation of heap-based vulnerabilities. HEAPX designs new path exploration strategies, a new memory model, and a new environment modelling solution so that the new enhanced symbolic execution engine can detect heap-based vulnerabilities and generate working vulnerabilities for them more efficiently.

Heap utilization primitives. Liu [47] uses pointer dependency analysis to identify key behaviours of primitives and then uses primitives to trigger vulnerabilities. Sean Heelan [48] proposed a method to automatically generate *Heap Overflow* vulnerabilities in the interpreter. By discovering the vulnerability exploitation primitives and determining the data structures that must be damaged, the attacker can achieve the desired exploitation method. The form of vulnerability that Heelan [49] primarily targets is out-of-bounds (OOB) reads of heap-related memory. HEAPHOPPER [50], proposes an automated method based on model checking and symbolic execution to analyze the exploitability of heap implementations in the presence of memory corruption. HEAPHOPPER will only predict the feasibility of exploitation, and through the verification of this feasibility, evaluate how the application of ptmalloc [35] will significantly weaken system security.

10. Conclusions

In this paper, we introduce the heap-related vulnerability and Heap-related exploit methods. The heap-related vulnerability has four types. At the same time, we introduce the primitive extraction and the exploit generation methods. Among the vulnerability detection methods, symbolic execution is selected to solve the condition of every branch. AAHEG can detect four heap-related vulnerability types and use AST to help generate the exploit of the corresponding vulnerability. AAHEG applies the advanced heap exploit method to generate the exploit. Experimental results show that AAHEG can complete 20 CTF problems' exploit generation, 11 of which have all protection mechanisms enabled.

Author Contributions: Conceptualization, Y.W. and Z.L.; methodology, Y.W. and Z.L.; software, Y.W.; validation, Y.W.; formal analysis, Y.W.; resources, Y.W. and Y.Z.; visualization, Y.W.; supervision, Y.W. and Y.Z.; project administration, Y.W. and Y.Z.; funding acquisition, Y.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Natural Science Foundation of China (Grant Nos. 62276017, U1636211, 61672081), the 2022 Tencent Big Travel Rhino-Bird Special Research Program, and the Fund of the State Key Laboratory of Software Development Environment (Grant No. SKLSDE-2021ZX-18).

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Song, J.; Alves-Foss, J. The DARPA Cyber Grand Challenge: A Competitor's Perspective. *IEEE Secur. Priv.* **2015**, *13*, 72–76. [CrossRef]
2. Huang, S.; Huang, M.; Huang, P.; Lai, C.; Lu, H.; Leong, W. CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations. In Proceedings of the Sixth International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, MD, USA, 20–22 June 2012; Volume 2012, pp. 78–87. [CrossRef]
3. Cha, S.K.; Avgerinos, T.; Rebert, A.; Brumley, D. Unleashing Mayhem on Binary Code. In Proceedings of the IEEE Symposium on Security and Privacy, SP 2012, San Francisco, CA, USA, 21–23 May 2012; pp. 380–394. [CrossRef]
4. Kc, G.S.; Keromytis, A.D. e-NeXSh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing. In Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005), Tucson, AZ, USA, 5–9 December 2005; pp. 286–302. [CrossRef]
5. Xu, S.; Wang, Y. BofAEG: Automated Stack Buffer Overflow Vulnerability Detection and Exploit Generation Based on Symbolic Execution and Dynamic Analysis. *Secur. Commun. Netw.* **2022**, *2022*, 1251987. [CrossRef]
6. Mow, W.; Huang, S.; Hsiao, H. LAEG: Leak-based AEG using Dynamic Binary Analysis to Defeat ASLR. In Proceedings of the IEEE Conference on Dependable and Secure Computing, DSC 2022, Edinburgh, UK, 22–24 June 2022; pp. 1–8. [CrossRef]
7. 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, CCS 2018, Toronto, ON, Canada, 15–19 October 2018; Lie, D., Mannan, M., Backes, M., Wang, X., Eds.; ACM: New York, NY, USA, 2018; pp. 1914–1927. [CrossRef]
8. 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 2021, Online, 11–13 August 2021; Bailey, M., Greenstadt, R., Eds.; USENIX Association: Berkeley, CA, USA, 2021; pp. 1647–1664.
9. Position-Independent Code. Available online: https://en.wikipedia.org/wiki/Position-independent_code (accessed on 30 October 2023).
10. Bierbaumer, B.; Kirsch, J.; Kittel, T.; Francillon, A.; Zarras, A. Smashing the Stack Protector for Fun and Profit. In Proceedings of the 24th IFIP World Computer Congress, WCC 2018, Poznan, Poland, 18–20 September 2018; Janczewski, L.J., Kutylowski, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2018; Volume 529, pp. 293–306. [CrossRef]
11. FULL RELRO. Available online: https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-onlyrelro (accessed on 30 October 2023).
12. Pwntools. CTF Framework and Exploit Development Library. 2020. Available online: https://github.com/Gallopsled/pwntools (accessed on 1 December 2023).
13. Wang, R.; Pan, Z.; Shi, F.; Zhang, M. Aemb: An automated exploit mitigation bypassing solution. *Appl. Sci.* **2021**, *11*, 9727. [CrossRef]

14. 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 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, 30 October–3 November 2017; Rosu, G., Penta, M.D., Nguyen, T.N., Eds.; IEEE Computer Society: Washington, DC, USA, 2017; pp. 274–279. [CrossRef]
15. Avgerinos, T.; Cha, S.K.; Rebert, A.; Schwartz, E.J.; Woo, M.; Brumley, D. Automatic exploit generation. *Commun. ACM* **2014**, *57*, 74–84. [CrossRef]
16. Huang, N.; Huang, S.; Chang, C. Analysis to heap overflow exploit in linux with symbolic execution. In *IOP Conference Series: Earth and Environmental Science*; IOP Publishing: Bristol, UK, 2019; Volume 252, p. 042100.
17. Zhao, Z.; Wang, Y.; Gong, X. HAEPG: An Automatic Multi-hop Exploitation Generation Framework. In Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment—17th International Conference, DIMVA 2020, Lisbon, Portugal, 24–26 June 2020; Maurice, C., Bilge, L., Stringhini, G., Neves, N., Eds.; Springer: Berlin/Heidelberg, Germany, 2020; Volume 12223, pp. 89–109. [CrossRef]
18. Sotirov, A. Heap feng shui in javascript. *Black Hat Eur.* **2007**, *2007*, 11–20.
19. Yun, I.; Kapil, D.; Kim, T. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. In Proceedings of the 29th USENIX Security Symposium, USENIX Security 2020, Boston, MA, USA, 12–14 August 2020; Capkun, S., Roesner, F., Eds.; USENIX Association: Berkeley, CA, USA, 2020; pp. 1111–1128.
20. Zhang, B.; Chen, J.; Li, R.; Feng, C.; Li, R.; Tang, C. Automated Exploitable Heap Layout Generation for Heap Overflows Through Manipulation Distance-Guided Fuzzing. In Proceedings of the 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, 9–11 August 2023; Calandrino, J.A., Troncoso, C., Eds.; USENIX Association: Berkeley, CA, USA, 2023; pp. 4499–4515.
21. Heelan, S.; Melham, T.; Kroening, D. Automatic Heap Layout Manipulation for Exploitation. In Proceedings of the 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, 15–17 August 2018; Enck, W., Felt, A.P., Eds.; USENIX Association: Berkeley, CA, USA, 2018; pp. 763–779.
22. Gennissen, J.; O'Keeffe, D. Hack the Heap: Heap Layout Manipulation made Easy. In Proceedings of the 43rd IEEE Security and Privacy, SP Workshops 2022, San Francisco, CA, USA, 22–26 May 2022; pp. 289–300. [CrossRef]
23. Li, R.; Zhang, B.; Chen, J.; Lin, W.; Feng, C.; Tang, C. Towards Automatic and Precise Heap Layout Manipulation for General-Purpose Programs. In Proceedings of the 30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, CA, USA, 27 February–3 March 2023.
24. Kang, X.; Debray, S. A Framework for Automatic Exploit Generation for JIT Compilers. In Proceedings of the Checkmate@CCS 2021, Research on offensive and defensive techniques in the Context of Man at the End (MATE) Attacks, Virtual Event, Republic of Korea, 19 November 2021; Hauser, C., Kwon, Y., Banescu, S., Eds. ACM: New York, NY, USA, 2021; pp. 11–19. [CrossRef]
25. Jin, L.; Cao, Y.; Chen, Y.; Zhang, D.; Campanoni, S. ExGen: Cross-platform, Automated Exploit Generation for Smart Contract Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2023**, *20*, 650–664. [CrossRef]
26. Krupp, J.; Rossow, C. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In Proceedings of the 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, 15–17 August 2018; Enck, W., Felt, A.P., Eds.; USENIX Association: Berkeley, CA, USA, 2018; pp. 1317–1333.
27. Huang, S.; Huang, M.; Huang, P.; Lu, H.; Lai, C. Software Crash Analysis for Automatic Exploit Generation on Binary Programs. *IEEE Trans. Reliab.* **2014**, *63*, 270–289. [CrossRef]
28. Jiang, Z.; Zhang, Y.; Xu, J.; Sun, X.; Liu, Z.; Yang, M. AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities. In Proceedings of the 44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, 21–25 May 2023; pp. 2122–2137. [CrossRef]
29. Wu, W.; Chen, Y.; Xu, J.; Xing, X.; Gong, X.; Zou, W. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In Proceedings of the 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, 15–17 August 2018; Enck, W., Felt, A.P., Eds.; USENIX Association: Berkeley, CA, USA, 2018; pp. 781–797.
30. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16–20 November 2020; Cohn, T., He, Y., Liu, Y., Eds.; Association for Computational Linguistics: Toronto, ON, Canada, 2020; Volume EMNLP 2020; pp. 1536–1547. [CrossRef]
31. Liguori, P.; Al-Hossami, E.; Cotroneo, D.; Natella, R.; Cukic, B.; Shaikh, S. Can we generate shellcodes via natural language? An empirical study. *Autom. Softw. Eng.* **2022**, *29*, 30. [CrossRef]
32. Yang, G.; Chen, X.; Zhou, Y.; Yu, C. DualSC: Automatic Generation and Summarization of Shellcode via Transformer and Dual Learning. In Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, 15–18 March 2022; pp. 361–372. [CrossRef]
33. Liguori, P.; Al-Hossami, E.; Orbinato, V.; Natella, R.; Shaikh, S.; Cotroneo, D.; Cukic, B. EVIL: Exploiting Software via Natural Language. In Proceedings of the 32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, 25–28 October 2021; Jin, Z., Li, X., Xiang, J., Mariani, L., Liu, T., Yu, X., Ivaki, N., Eds.; IEEE: Piscataway, NJ, USA, 2021; pp. 321–332. [CrossRef]
34. Yang, G.; Zhou, Y.; Chen, X.; Zhang, X.; Han, T.; Chen, T. ExploitGen: Template-augmented exploit code generation based on CodeBERT. *J. Syst. Softw.* **2023**, *197*, 111577. [CrossRef]

35. Gloger, W. Ptmalloc. 2006. Available online: https://github.com/hustfisher/ptmalloc/blob/master/malloc.c (accessed on 1 December 2023).
36. Linux Manual Page. 2023. Available online: https://man7.org/linux/man-pages/man2/syscalls.2.html (accessed on 1 December 2023).
37. MaXX. Vudo—An Object Superstitiously Believed to Embody Magical Powers. 2001. Available online: http://phrack.org/issues/57/8.html (accessed on 1 December 2023).
38. Once upon a Free(). 2001. Available online: http://phrack.org/issues/57/9.html (accessed on 1 December 2023).
39. Mandt, T. Kernel Pool Exploitation on Windows 7. 2011. Available online: https://media.blackhat.com/bh-dc-11/Mandt/BlackHat_DC_2011_Mandt_kernelpool-wp.pdf (accessed on 1 December 2023).
40. Karimi, A. A Survey of Heap-Exploitation Techniques. 2021. Available online: https://www.researchgate.net/profile/Alireza-Karimi-31/publication/369594354_A_survey_of_heap-exploitation_techniques/links/6423d78392cfd54f84388e5b/A-survey-of-heap-exploitation-techniques.pdf (accessed on 1 December 2023).
41. david942j. One_Gadget. 2023. Available online: https://github.com/david942j/one_gadget/releases (accessed on 1 December 2023).
42. Phantasmagoria, P. The Malloc Maleficarum. *Bugtraq Mailinglist* 2005. Available online: https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt (accessed on 1 December 2023).
43. Wang, F.; Shoshitaishvili, Y. Angr—The Next Generation of Binary Analysis. In Proceedings of the IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, 24–26 September 2017; pp. 8–9. [CrossRef]
44. Radareorg. Radare2. 2023. Available online: https://github.com/radareorg/radare2 (accessed on 1 December 2023).
45. Ctftime. Available online: https://ctftime.org/ (accessed on 1 December 2023).
46. Tu, H. Boosting Symbolic Execution for Heap-based Vulnerability Detection and Exploit Generation. In Proceedings of the 45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, 14–20 May 2023; pp. 218–220. [CrossRef]
47. Liu, J.; An, H.; Li, J.; Liang, H. Detecting Exploit Primitives Automatically for Heap Vulnerabilities on Binary Programs. *arXiv* **2022**, arXiv:2212.13990. [CrossRef]
48. Heelan, S.; Melham, T.; Kroening, D. Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, 11–15 November 2019; Cavallaro, L., Kinder, J., Wang, X., Katz, J., Eds.; ACM: New York, NY, USA, 2019; pp. 1689–1706. [CrossRef]
49. Heelan, S.; Melham, T.; Kroening, D. Heap Layout Optimisation for Exploitation (Technical Report). Available online: https://www.blackhat.com/docs/eu-17/materials/eu-17-Heelan-Heap-Layout-Optimisation-For-Exploitation-wp.pdf (accessed on 10 December 2023).
50. Eckert, M.; Bianchi, A.; Wang, R.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security. In Proceedings of the 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, 15–17 August 2018; Enck, W., Felt, A.P., Eds.; USENIX Association: Berkeley, CA, USA, 2018; pp. 99–116.