HAIT: Heap Analyzer with Input Tracing

Andrea Atzeni[1], Andrea Marcelli[1], Francesco Muroni[2] and Giovanni Squillero[1]
[1]*DAUIN, Politecnico di Torino, Corso Duca degli Abruzzi 24, Torino, Italy*
[2]*Independent Scholar, Torino, Italy*

Keywords: Heap, Exploit, Memory Profiler, Dynamic Symbolic Execution, Taint Analysis.

Abstract: Heap exploits are one of the most advanced, complex and frequent types of attack. Over the years, many effective techniques have been developed to mitigate them, such as *data execution prevention*, *address space layout randomization* and *canaries*. However, if both knowledge and control of the memory allocation are available, *heap spraying* and other attacks are still feasible. This paper presents HAIT, a memory profiler that records critical operations on the heap and shows them graphically in a clear and comprehensible format. A prototype was implemented on top of *Triton*, a framework for dynamic binary analysis. The experimental evaluation demonstrates that HAIT can help identifying the essential information needed to carry out heap exploits, providing valuable knowledge for an effective attack.

1 INTRODUCTION

An *exploit* is a combination of code and data that takes advantage of a flaw in a system to cause an unintended behavior, for instance, allowing to gain illegitimate control over it. Memory exploits can target different regions, namely the stack or the heap, and can adopt different methods to circumvent operating system (OS) defenses.

The development of an exploit consists of two phases: the discovery of the vulnerability and the design of the code to take advantage of it. Manual vulnerability discovery is typically not an option since too complex and time consuming. In order to find an unexpected behavior, the target system can be modeled (e.g. through *symbolic execution*), or stressed with automatically generated inputs (e.g. by *fuzzing*). Eventually, the development of the exploit leverages the flaw discovered in the code, transforming the system weakness into a concrete attack.

Buffer overflow attacks are one of the longest-running, occurring, and damaging type of threats (UKessays.com, 2015). In short, a buffer overflow exists whenever a program attempts to put more data in a buffer than it can actually hold. This situation can simply crash the component, or, more interestingly, can be used to execute arbitrary code, thus gaining the control over a service (MITRE, 2017). Roughly speaking, two kinds of buffer overflow attacks exist: *stack-based* and *heap-based*.

As a result of the progress in secure coding, in static and dynamic application analysis and in OS-level protection mechanisms, the recent attack trends show that stack-based exploitation is becoming less frequent in modern systems. On the other hand, the heap is the most targeted element of software processes (Rains, 2014): it can be either the core of the vulnerability or just an auxiliary element in a more complex exploit[1]. Corrupting the heap requires a large amount of information and, due to the number of variables and details involved, is much harder than stack-based attacks. As the level of complexity of the systems grows, developing a heap exploit by manually inspecting all the memory becomes less feasible and ultimately depends on the analyst experience and ability.

In order to discover the essential pieces of information required for the heap exploitation, we surveyed different well-known techniques, either generic, such as *use-after-free* and *double-free*, or tailored to a specific allocator. Our research clearly showed that all the attacks require two categories of information: the list of changes in the layout of the heap memory during execution and the knowledge of how the program inputs influence the operations on the heap. Since many attacks rely on a specific memory layout, it is crucial to know the heap state at each step of the execution, i.e. the exact position of

[1]Like heap spraying in ASLR circumvention, quite a common strategy in browser exploitation

Atzeni, A., Marcelli, A., Muroni, F. and Squillero, G.
HAIT: Heap Analyzer with Input Tracing.
DOI: 10.5220/0006420803270334
In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT*, pages 327-334
ISBN: 978-989-758-259-2

memory blocks, whether they are currently allocated and all the associated metadata. Considering the high level of sophistication of the attacks, a graphical representation of the heap layout at each step is very useful for the development of the exploit. Additionally, the developer needs to understand how the program inputs influence the heap layout, which is essential to decide the exploitation technique to be used.

In this paper we present a methodology to automatically gather information about the heap state and the operations that are performed on it. While the idea is general and can be applied to different architectures and applications, the research tackled the exploit development on Linux desktop applications.

We implemented the proposed methodology in a proof-of-concept tool named HAIT (Heap Analyzer with Input Tracing). The prototype is built on top of Triton (Saudel and Salwan, 2015), a framework for the dynamic binary analysis of programs. Triton consists of different components, among which there is a *dynamic symbolic execution engine* that is the foundation of the proposed tool. For the dynamic binary instrumentation, our prototype relies on Pin[2], which offers a good integration with Triton.

HAIT has been successfully tested on several Capture-The-Flag challenges (CTFs), security competitions where purposely vulnerable programs are used for practicing with various security related challenges. Our tool provided important support and it was able to significantly speed-up the information gathering, one of the crucial phase of the heap exploitation process, relieving the users from a long and tedious manual inspection.

The rest of the paper is organized as follows: Section 2 provides the necessary background, Section 3 introduces the proposed methodology and HAIT, while Section 4 presents an example case of study. Finally, Section 5 concludes the paper.

2 BACKGROUND

2.1 Automated Vulnerability Analysis

The development of an exploit is characterized by the discovery of a vulnerability and the design of the code to take advantage of it. Regarding the discovery, most of the automated vulnerability analysis systems can be classified into three categories: *static*, *dynamic*, and *concolic*; each one with its typical advantages and disadvantages (Stephens et al., 2016). Static analysis is performed without actually executing programs: it analyzes the control and data flows, builds a model, and checks its properties. While it is usually quite fast and able to produce deterministic results, its performance depends entirely on the choice of the model: its complexity may lead to intractable problems, or it may simulate only a subset of the program features. Moreover, static analyses provide a significant fraction of false positive alerts. On the contrary, dynamic analyses monitor the native execution of the applications in specific conditions, such as with random inputs generated by a *fuzzer*. The dynamic analysis does not suffer from the problems of the static one, but the time required depends on the size and complexity of the test, and its final accuracy is hardly quantifiable. Finally, concolic execution engines make use of program interpretation and constraint-solving techniques to generate inputs able to explore the state space, in an attempt to reach and trigger vulnerabilities. However, due to the large number of paths that are executed, these systems are not scalable.

Other types of analysis, which could be either defined as static or dynamic, also exist. For instance, *Taint analysis* is an iterative process where the goal is to eventually mark locations as tainted if those derive directly from relevant sources (e.g. user input) (Heelan, 2009). Taint analysis is a powerful methodology, and in HAIT the same principle is used to correlate program input to memory allocation in order to find allocation patterns that can be used to create the memory layout of interest.

2.2 Exploitation Techniques

Early buffer overflow exploits relied on the ability to inject executable code, termed *shellcode*, into a buffer and then overwrite a return address on the stack to point to it. As a consequence, instead of returning to the previous code, execution would jump into the shellcode, giving the attacker control over the program. Security researchers dealt with the problem by preventing the stack and other memory areas not supposed to contain code from being executed. Microsoft introduced Data Execution Prevention (DEP) in Windows XP, marking memory as non-executable with the NX bit, if available; around the same time, OpenBSD implemented a feature named W$\oplus$X, which forces each memory page to be either writable or executable, but not both. To outsmart such defenses, attackers adopted the idea of code reuse: take advantage of legitimate functionality already in the program to accomplish their malicious goals. For instance, in the return-into-libc exploits, an attacker

[2]Pin, a dynamic binary instrumentation tool (v. 3.2), https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

redirects the control flow directly to a sensitive libc function, such as `system()`, after setting the proper arguments. Another possibility to circumvent non-executable memory is to hijack program control flow and then execute chosen instruction fragments that are already present in the machine's memory. Such fragments, or *gadgets*, are typically located at the end of a subroutine and conclude with a return instruction, hence the name Return-Oriented Programming (ROP) (Wojtczuk, 2001; Roemer et al., 2012). Against such attacks, researchers developed a system-level hardening technique called Address Space Layout Randomization (ASLR) (Szekeres et al., 2013): the programs memory layout, including the locations of libraries, the stack, and the heap, is randomized at each execution. Thus, with ASLR, the attacker does not know where to redirect the control flow in the libraries to execute specific functions. Moreover, even if the attacker can determine this information, he would be still unable to identify the location of specific functions inside the library unless in possession of a copy of the library itself. As a result, an attacker usually has to provoke the library content leakage and parse the code to identify the location of critical functions.

Looking for a good memory corruption, i.e. that allow an attacker to execute a "reliable exploit", often the *unlink* procedure of the heap allocator is the target of the attack, as described in the well-known *Malloc Maleficarum* Phrack Magazine articles (Phantasmagoria, 2005; blackngel, 2009). Both the double-free and the one-byte-overflow vulnerabilities (Conrad, 2015), in which chunk metadata is overwritten or emulated, allow to achieve a *write-primitive*, that is the possibility to write "where you want, whatever you want". Moreover, use-after-free vulnerabilities can also lead to very reliable exploits (Evans, 2015), particularly when the "free" and the "use" are close together (e.g., in the Pinkie Pie[3] exploit). In real-world exploits, an attacker often uses an information disclosure attack to leak the address or contents of a library, then uses this information to calculate the correct address of a security-critical library function (such as `system()`), and finally sends a second payload to the vulnerable application that redirects the control flow to call the desired function (Di Federico et al., 2015).

2.3 Related Tools

Shadow (Argyroudis, P. and Karamitas, C., 2015) is a Python extension for WinDBG that provides an extremely detailed insight of every aspects of *jemalloc*, the heap allocator of Firefox. Designed for Windows, it allows to extract several meta-information about *jemalloc* and to display Firefox symbols, retrieved from Mozilla symbol server.

Villoc[4] is a tool designed for the visualization of the heap memory layout. The program under analysis is executed within *ltrace*, the Linux library call tracer. Then, the output is parsed by a Python script which looks for calls to heap management functions and finally produces a static HTML file with a graphical representation of the different states of the heap after each function call. Villoc main limitation is that the analysis is only available when the program under analysis terminates, so there is no immediate feedback at runtime. Moreover, since the information displayed is solely based on *ltrace*, other valuable data, e.g. chunk metadata, is not accessible.

HAIT extends the idea of *Villoc* using an underlying powerful *dynamic binary analysis framework*, such as Triton. The proposed tool not only display the crucial chunk metadata, although currently only *ptmalloc* is implemented, but it also allows a step by step analysis correlating program inputs to heap allocations.

3 PROPOSED FRAMEWORK

Any analysis of heap exploitation methodologies (Shoshitaishvili et al., 2016), and real-word example show that all attacks require two categories of information: the knowledge on how the heap memory layout changes during program execution and how user inputs influence the allocations on the heap.

To support the information gathering of the crucial knowledge of every exploit, we introduce HAIT[5], a proof of concept Heap Analyzer with Input Tracing. The tool is built on top of Triton (Saudel and Salwan, 2015), a sophisticated framework for *dynamic binary analysis*, which consists of several components, above all a *dynamic symbolic execution engine*.

Developing HAIT, we took the same approach as Automatic Exploit Generation (Avgerinos et al., 2011): leverage binary instrumentation to obtain data as soon as it is available, a technique whereby extra code is injected into the normal execution flow, therefore allowing an arbitrary analysis of the executing program (Heelan, 2009). Binary instrumentation exists in two flavors: *static* and *dynamic*. In the first, the additional code is added at compile time,

[3]http://scarybeastsecurity.blogspot.it/2013/02/exploiting-64-bit-linux-like-boss.html

[4]Villoc, https://github.com/wapiflapi/villoc/

[5]https://github.com/mauronz/HAIT

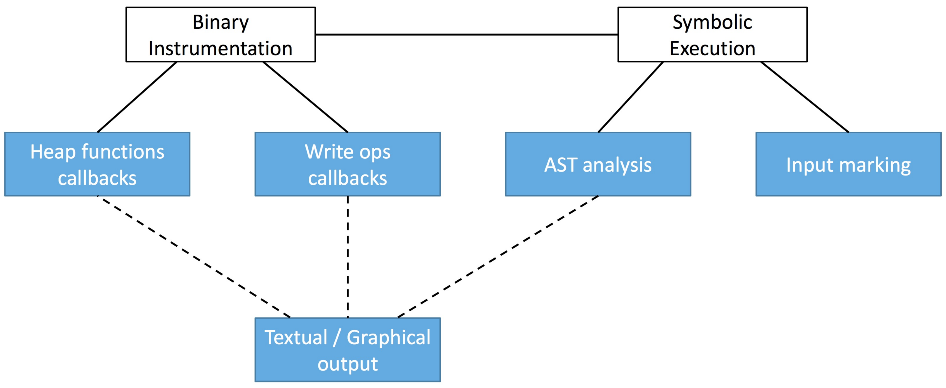


Figure 1: HAIT schema.

resulting in a new version of the original executable. The main drawback of this technique is the need of the application source code. On the other hand, *dynamic* binary instrumentation works directly on the executable, looking for events that trigger specific routines. Considering the purpose of tracing heap operations, we decided to use the *dynamic* approach. As a tracer, HAIT relies on Pin, a dynamic binary instrumentation library from Intel, which currently offers the best integration with Triton, since the framework provides Python bindings to interact directly with the tracer and use all of its features, above all event hooking. Figure 1 illustrates the HAIT infrastructure.

By running specific routines both before and after the execution of all the heap related functions, HAIT can perform a detailed analysis. Specifically, it retrieves the calling parameters, either from the stack or the registers, as defined by the calling convention of Linux 64-bit executables. Moreover, it inspect the memory to get the chunk metadata; these can be found at predetermined offsets with respect to the address of the chunk, i.e. the address returned by the allocation functions. The offsets are computed from the corresponding data structure of the *ptmalloc* implementation (*struct malloc_chunk*). Tracking program writes to the heap, by means of Pin function hooks, provides additional useful information.

Symbolic expressions are stored in the form of Abstract Syntax Trees (AST), a binary tree data structures in which each node represents an operation and the two children the operands, while taint analysis is used to keep track of user input affected allocations.

HAIT logs memory allocations, write operations and tainted values to the console; Section 4 provides several examples. Then, as illustrated in Figure 3, an interactive HTML page is built, displaying at each line the current heap memory layout. The white background color identifies a free memory block, while a red border is drawn if an allocation is controlled by user input. By clicking a memory chunk, it is possible to retrieve the memory addresses and other allocator-specific metadata, such as *fd* and *bk* links in the case of a freed block. Moreover, additional information from */proc/pid/maps* is displayed. Such features have been shown to be crucial for the fast prototyping of heap exploit.

Table 1: Memory usage comparison for several CTF[6].

	Memory usage (GiB)	
	with *libc*	without *libc*
Freenote	2.3	0.40
Stkof	2.0	0.04
Logger	2.4	0.08
Chat	2.1	0.10
Shopping	2.3	0.07

As any other analysis tool, using HAIT implies overhead to program execution. By default *dynamic binary instrumentation* increases the time of the execution, besides considering that the several analyses carried during program execution, the overhead can reach up to 500%.

This cost is mostly expected and in line with other analysis performed using the Triton framework [7].

There exist different ways to improve the tool, as discussed in Sec. 5, and thus enlarge its applicability.

Table 1 shows a comparison of RAM memory usage while analyzing several CTF programs. It is clear that including the *libc* library in program analysis causes a big overhead in term of RAM memory allocation. While it is possible to remove it without affecting HAIT analysis, it provides critical information to link the program inputs to the heap allocations.

However, our tool is meant to be applied mostly *after* the vulnerability discovery phase, i.e. to de-

[7]Triton's author states "*By default DBI (Dynamic binary instrumentation) increases the time of the execution. Add others analysis and you got an overhead of 500% to 1000%*" (Salwan, 2015)

velop a targeted exploit. Thus, HAIT is not meant to be generically applied on large binaries, like Firefox and Chrome, for which would be unpractical. On the contrary, should target specific parts (e.g. identified libraries) where vulnerabilities exist (or are suspected to exist). In this kind of scenarios our tool is effective: we tested HAIT in the context of several CTFs (i.e. where programs vulnerable by design are provided for security analysis) and it proved to be very useful, as detailed in Sec.4.

4 CASE OF STUDY

As a demonstration of the capabilities of HAIT, we selected *freenote*[8], a Capture The Flag (CTF) challenge from the 0CTF Quals 2015 (Quals, 2015). The vulnerable program is an implementation of a Linux command-line textual notebook, where a user can add, modify or delete a note, as well as print all the previously inserted ones. The following analysis of the vulnerable program is based on the write-up of a possible solution provided by one of the competitors (seanwupi, 2015).

Freenote is a typical CTF designed to challenge reversers to develop a working heap exploit with all the operating system countermeasures active, like ASLR, DEP and partial RELRO. The challenge is provided as an executable program, in ELF format.

The following description will focus on the initial information gathering process about the heap state and the data structures, aiming at the leakage of a memory address to bypass ASLR. The details about how to take advantage of such information and the development of the actual exploit are omitted, pointing the attention on the support that HAIT provides.

The faulty program presents two vulnerabilities that can be targeted by an attacker: a null-terminated-string and a double free. When a new note is inserted, the user input is written in memory without appending the ASCII null character '\0', therefore, when the *printf* function is invoked, more information than the necessary will be presented to the user. On the other hand, when a note is deleted, the program wrongly manage the pointer to the allocated area where the note was stored, resulting in a double free vulnerability [9]. While both bugs can be identified reversing the code by an expert and trained eye, HAIT presents them much more readily.

[8]https://github.com/ctfs/write-ups-2015/tree/master/0ctf-2015/exploit/freenote

[9]https://www.owasp.org/index.php/Double_Free

4.1 *Freenote* Analysis with HAIT

The first step of the analysis is to gather as much information as possible about the program under study, it took few seconds and required about 2 GB of RAM. By running the executable inside HAIT, the following output is shown.

```
[*] Malloc -> addr = 0x8c3008, usize =
    0x1810, rsize = 0x1820
```

A block of 6160 (0x1810) bytes is allocated as soon as *Freenote* starts. 16 additional bytes are reserved at the beginning of the block for storing metadata information from *ptmalloc*.

Then, the user can choose the action to perform. Adding a new note, *option 2*, produces the following output.

```
Your choice: 2 - Length of new note: 10
[*] Malloc -> addr = 0x8c4828, usize =
    0x80, rsize = 0x90
SymVar_0 read #1 byte 0 -> value='2'
    -> atoi
SymVar_2 read #3 byte 0 -> value='1'
    -> atoi
SymVar_3 read #4 byte 0 -> value='0'
    -> atoi
```

Even though the user inserts a length of ten characters, a block of 128 (0x80) bytes is allocated. Trying different sizes, it is trivial to deduce that the length provided by the user is rounded up to the next multiple of 128 bytes.

As shown by the above *read* operations, thanks to our tracing system, HAIT can correlate user input to memory allocation without requiring the reverser any further manual analysis.

Finally, the content of the note is actually stored, after which HAIT outputs several logs. An example snippet is shown in the following listing.

```
[*] Continued write operation in block
    0x8c3008, starting at 8c3018
    (current size 0x18)
[*] Write operation in block 0x8c3008,
    at 0x8c3010 (size 0x8) value=0x1
```

In the first operation, 24 (0x18) bytes are written at offset 16 (0x10), in the big block allocated during initialization. In the second, the integer value '1' is written at offset 8. The meaning of this operations become clear after trying to add several notes. The first write always occurs at an offset calculated according to the following formula:

$$offset_i = 16 + 24 \cdot i$$

Table 2: Note index entry structure.

8 byte	8 byte	8 byte
Unknown	Length of note	Pointer of memory block of the note

where *i* is the note number, with a 0-based notation. Differently, the second write is always at the same location and it is legitimate to infer that it is a value that keeps track of the number of notes.

When *option 4* is chosen, the selected note is deleted. As shown by the following output, three actions occur. Firstly, the number of allocated notes is decremented, then the note entry in the index is updated and finally, the memory chunk containing the note is freed.

```
Your choice: 4 - Note number: 0
[*] Write operation in block 0x8c3008,
    at 0x8c3010 (size 0x8) value=0x1
[*] Continued write operation in block
    0x8c3008, starting at 0x8c3018
    (current size 0x10)
[*] Free -> addr = 0x8c4828
```

To conclude the analysis, the update of an existing note, *option 3*, is shown.

```
Your choice: 3
Note number: 1
Length of note: 10
...
[*] Realloc -> addr = 0x8c48b8, usize
    =  0x80, rsize = 0x90 (prev block
    = 0x8c48b8)
...
[*] Write operation in block 0x8c3008,
    at 0x8c3048 (size 0x8)
    value=0x8c48b8
[*] Write operation in block 0x8c3008,
    at 0x8c3040 (size 0x8) value=0xa
```

The update operation is implemented with a *realloc* and thanks to the logs from HAIT it is possible to deduce that the first write stores the address of the re-allocated memory block at offset 16 (0x10), while the second one writes the size of the note at offset 8. Thanks to the previous analysis, it is possible to infer that the note entry in the index data structure consists of 8 unknown bytes, 8 bytes where the size of the note is stored and 8 bytes with the address of the memory block containing the note itself, as illustrated by Table 2.

Having gathered such information about the data structure was essential to discover the double-free vulnerability: recalling the penultimate listing, when a note is deleted only the first 16 bytes are updated, that is, the pointer to the chunk containing the note is not removed from its entry in the index.

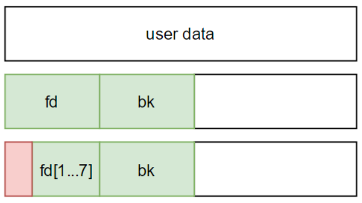


Figure 2: Leak of *fd* link.

4.2 Address Leak

As discussed in Section 2.2, Address Space Layout Randomization prevents to trivially use any predetermined memory address while taking advantage of a vulnerability. This result is achieved by the randomizing the offset at which each section of the executable is mapped at each execution of the program. To successfully create a working exploit, both the address of the heap and *libc* must be leaked. Those will be later used in the writing of the actual exploit.

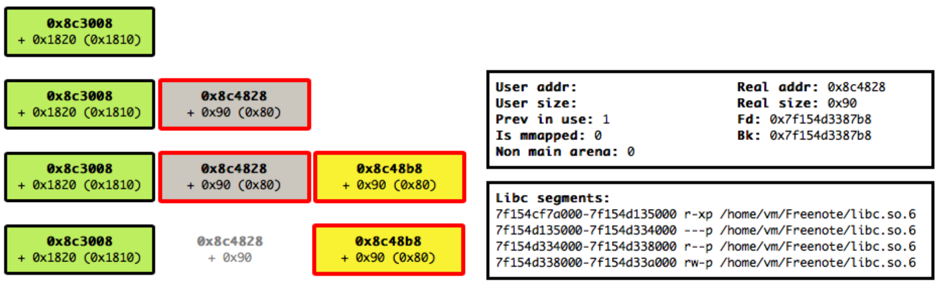
The procedure to leak the address of *libc* is the following:

1. Allocate two notes (0 and 1)
2. Free the first one (note 0)
3. Allocate a new note on top of the first one with size 1
4. Print all the current notes

That specific sequence of actions targets the implementation of *free* in *ptmalloc*: once a chunk of memory is freed, it is inserted in a *bin*[10] of free blocks. Being the first memory chunk to be released, *backwork* (*bk*) and *forward* (*fd*) links will point to the head of the list, which is always located in the same position inside the *.bss* section of the *libc* image. Referring to Figure 2, since the *forward* link is located at the beginning of memory chunk, allocating a new block of one-byte length will result in overwriting only the first byte of the *fd* link. By printing all the notes, the faulty program will show the leaked addresses inside the *libc* image. The initial creation of two notes was a necessary step to have a free list. Otherwise, the freed block, corresponding to note 0, would have been merged with the subsequent non-allocated memory space.

Referring to Figure 3, looking at the details of the free chunk, it is possible to confirm that the memory

[10]A *bin* is a double linked list of free chunks carefully tracked by the allocator for an efficient reuse. *Forward* and *backword* links are part of the chunk metadata.



```
User addr:                         Real addr: 0x8c4828
User size:                         Real size: 0x90
Prev in use: 1                     Fd: 0x7f154d3387b8
Is mmapped: 0                      Bk: 0x7f154d3387b8
Non main arena: 0
```

```
Libc segments:
7f154cf7a000-7f154d135000 r-xp /home/vm/Freenote/libc.so.6
7f154d135000-7f154d334000 ---p /home/vm/Freenote/libc.so.6
7f154d334000-7f154d338000 r--p /home/vm/Freenote/libc.so.6
7f154d338000-7f154d33a000 rw-p /home/vm/Freenote/libc.so.6
```

Figure 3: HTML view of the leaking procedure.

address stored in *fd* and *bk* links is inside *libc* memory area: they have the same value and the address *0x7f154d3387b8* belongs to the first *libc* segment as shown in the memory mapping from */proc/pid/maps*.

The leak of the heap address is conceptually similar to the one of *libc* and for the sake of brevity, it will not be shown.

4.3 Analysis of the Results

Despite the relative simplicity of the program under analysis, HAIT showed several crucial capabilities required by a tool in the arsenal of an exploit writer: a fast prototyping, ready visualization of essential information (e.g. memory addresses and program flaws) and easy debug support.

Referring to Figure 3, the tool readily showed the memory address of the heap and *libc* library, allowing fast exploit prototyping. Although those addresses vary at each program execution, it allows the reveser to focus on the exploit itself, leaving the information leakage to further analyses. Having a visual representation of the heap memory state at each step of the execution, using the HTML view, is much more human intuitive than reading a collection of hexadecimal addresses. Moreover, both the two core vulnerabilities can be easily spotted by carefully reading the few output lines. Finally, the exploit writing process itself is further supported by a handy debug, thanks to the graphical visualization and by a step-by-step analysis.

The previous example showed the level of sophistication that an average heap exploit requires. Although experience and technical knowledge are not to be questioned and represent the foundation for taking advantage of any vulnerability, the availability of such a tool represents a considerable support to the exploit development process. It is no exaggeration to say that HAIT is the Swiss-Army knife of heap exploitation.

5 CONCLUSIONS

Developing a successful exploit from scratch for an unknown application is a complex process. Twenty years ago, a simple buffer overflow, redirecting the execution to the attacker shellcode, was enough to hijack the control flow of a vulnerable program. Without any further security measure, such an exploit could be easily created using few lines of code after a simple manual analysis and testing.

Nowadays the increasing awareness of the importance of secure coding and the availability of several testing tools avoid trivial bugs in the code. Moreover, the introduction of several protection techniques at the operating system level, with the aim of increasing the global system security, make impossible to take advantage of most of the flaws of a target program. If this approach continues, one day the exploit development process will be so long and tedious to become unfeasible.

In such a complex environment, it is clear that security researchers need the support of specialized tools to gather as much information as possible to develop sophisticated exploits.

The research presented in this paper focuses on a specific category of exploits, those based on the heap: the most common target in modern exploitation. The heap can be the core of the vulnerability, like in those techniques that rely on the the heap corruption, or it can be just a part of a wider process, like the bypass of ASLR and DEP by means of heap spraying, which is the commonly used approach in almost all browser exploits. Considering the complexity of heap management, having a tool that automatically provides all the useful information, instead of manually retrieving them by inspecting the memory, can be the ultimate advantage for the security analysts to discover quicker an exploitable flaw and fix it.

The in-depth analysis of commonly used techniques has led to the definition of a summary about which is the most important kind of information to create heap-based exploits. This paper shows a methodology to obtain such data during the execution of the target program, taking advantage of dynamic binary instrumentation to perform a runtime analysis of the heap state. HAIT, the proof of concept implementation of our methodology, proved to be useful in the context of known vulnerable programs, like CTFs.

6 FUTURE DEVELOPMENT

HAIT has been developed to showcase the proposed exploitation methodology and is still not a production ready tool. As such, can be improved regarding coverage and effectiveness. The overhead can be reduced by creating an ad-hoc engine for the concolic execution, eliminating the unnecessary operations that the underlying generic framework, Triton, provides and focusing only to what strictly required by our analysis. Moreover, since the method of analysis is general and can be applied to a large variety of targets, it would be interesting to extend the tool to support other architectures and allocators, above all the Android environment, which runs on ARM and uses jemalloc for the heap management.

ACKNOWLEDGMENT

Andrea Marcelli Ph.D. program at Politecnico di Torino is supported by a fellowship from TIM (Telecom Italia Group).

Authors wish to thanks Dario Lombardo and Mariano Graziano for their support and insightful comments.

REFERENCES

Argyroudis, P. and Karamitas, C. (2015). Shadow v1.0b. https://github.com/CENSUS/shadow.

Avgerinos, T., Cha, S. K., Hao, B. L. T., and Brumley, D. (2011). AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium*, pages 283–300.

blackngel (2009). Malloc des-maleficarum. *Phrack*.

Conrad, E. (2015). Off by 1 overflow. https://www.sans.edu/student-files/presentations/heap_overflows_notes.pdf.

Di Federico, A., Cama, A., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. (2015). How the elf ruined christmas. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 643–658.

Evans, C. (2015). What is a good memory corruption. https://googleprojectzero.blogspot.it/2015/06/what-is-good-memory-corruption.html.

Heelan, S. (2009). Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford.

MITRE (2017). Common Weakness Enumeration - version 2.10, cwe-120: Buffer copy without checking size of input ('classic buffer overflow'). http://cwe.mitre.org/data/definitions/120.html.

Phantasmagoria, P. (2005). The malloc maleficarum - glibc malloc exploitation techniques. *Phrack*.

Quals (2015). freenote 0ctf. https://ctf.0ops.net.

Rains, T. (2014). How vulnerabilities are exploited: the root causes of exploited remote code execution cves.

Roemer, R., Buchanan, E., Shacham, H., and Savage, S. (2012). Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2.

Salwan, J. (2015). presentation of dynamic behavior analysis using binary instrumentation. https://www.slideshare.net/sth4ck/st-hack2015-dynamicbehavioranalysisusingbinaryinstrumentation-jonathansalwan-46443521.

Saudel, F. and Salwan, J. (2015). Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC.

seanwupi (2015). Advanced heap exploitation: 0ctf 2015 'freenote' writeup. https://gist.github.com/seanwupi/929df6655f2acdbab3ff.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. (2016). SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. (2016). Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*.

Szekeres, L., Payer, M., Wei, T., and Song, D. (2013). Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE.

UKessays.com (2015). Buffer Overflow Attacks And Types Computer Science Essay. https://www.ukessays.com/essays/computer-science/buffer-overflow-attacks-and-types-computer.science-essay.php.

Wojtczuk, R. (2001). The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*.