2015 IEEE 39th Annual International Computers, Software & Applications Conference

Reversing and identifying overwritten data structures for memory-corruption exploit diagnosis

Lei Zhao*†, Run Wang*†, Lina Wang*†, and Yueqiang Cheng‡
*Computer School of Wuhan University, Wuhan, China
†Key Laboratory of Aerospace Information Security and Trust Computing, Wuhan, China
‡CyLab, Carnegie Mellon University

***Abstract*—Exploits diagnosis requires great manual effort and desires to be automated as much as possible. In this paper, we investigate how the syntactic format of program inputs, as well as reverse engineering of data structures, could be used to identify overwritten data structures, and propose a binary-level exploit diagnosis approach, `deExploit`, that is generic to attack types and effective in identifying key attack steps. In details, we design to use a fine-grained dynamic tainting technique to model how the exploit is dynamically processed during program execution, dynamically reverse corresponding data structures of program input and then identify overwritten data structures by detecting the *deviation between dynamic processing of exploit and that of benign input*. We implement `deExploit` and perform it to diagnose multiple exploits in the wild. The results show that `deExploit` works well to diagnose memory corruption exploits.**

Keywords*—*software vulnerability; exploit diagnosis; memory corruption; reversing engineering

I. Introduction

Memory corruption exploit is one of the most serious problems in software security [1], [2]. For emergency response to such attacks, researchers have proposed many techniques to protect system from being exploited, such as vulnerability signature [3] and input filtering [4]. Among these techniques, exploit diagnosis is a fundamental progress to identify vulnerability root causes and generate vulnerability signatures. Unlike developers who could debug and fix memory corruption vulnerabilities with the support of source code, security analysts always perform exploit diagnosis at the binary level, thus critically need automated exploit diagnosis techniques to detect the attack and understand how the exploit works.

Exploit diagnosis involves at least two steps, attack detection and key attack steps identification. Most previous techniques mainly focus on attack detection. However, key attack steps identification is not less significant than attack detection. Specially, with the enhancement of security mechanisms like stack protection [5], address-space-layout randomization (ASLR) [6], and Data Execution Protection (DEP), memory corruption exploits become more and more sophisticated [1], [7] and always involve several attack steps to evade above protections. Identifying key attack steps and understanding how the exploit works are also desirable for many security applications such as digital forensics.

All correspondence should be addressed to Dr. Lei Zhao at Computer School, Wuhan University. Email: leizhao@whu.edu.cn.

In past decades, many techniques have been proposed to defend against memory corruption vulnerabilities [8], [9], [5], [10] (as shown in Table I). Software protection techniques, such as data-flow-integrity (DFI) [10] and WIT [8], require the support of source code. At the binary level, techniques like MemCheck [11] and Clause2007 [12] require the support of debugging information. However, exploit diagnosis is always performed on binary executables that are not protected by any protection techniques, or there is no debugging information, thus the above techniques may not be practical to executables without source code such as COTS binaries and legacy code.

There are other binary level techniques for attack detection, such as stack guard [5], dynamic tainting [13], and control-flow integrity [15]. There are two challenges for these attack detection techniques to diagnose exploits. For one thing, most of these techniques are designed based on the signature of attack manifestations [16] or illegal use of unsafe data [17], such as overwriting return addresses or function pointers. However, attack manifestations are limited by specific attack types. For example, TaintCheck [13] and control-flow integrity [15] are effective in detecting control-hijacking attacks, but fail to capture the attack manifestation of non-control data attack [18]. Therefore, an effective exploit diagnosis should be generic to attack types.

For another, most of previous detection techniques cannot identify key attack steps. Dependency-based slicing can be used to track contexts to help attack diagnosis [13], but dependency-based techniques cannot clearly distinguish key attack steps and other dependencies. PointerScope [1] aims at identifying key attack steps by detecting pointer misuses, however may lose its ability for some non-control data attacks ($\times^*$ as shown in Table I). For example, a pointer with the type of data pointer could be misused as another pointer with the same type during the non-control data attack, then no type conflict is detected.

Typical memory corruption exploits always involve overwriting program data structures as key attack steps. To detect overwritten data structures seems to be a promising approach for exploit diagnosis. In this paper, we investigate how the syntactic format of program input, as well as reverse engineering of data structures, could be used to identify overwritten data structures, and propose a binary-level exploit diagnosis approach named `deExploit` that is generic to attack types and effective in identifying key attack steps.

Program inputs usually contain structural information, which is often denoted as *syntactic format* [19], [20]. When

0730-3157/15 $31.00 © 2015 IEEE
DOI 10.1109/COMPSAC.2015.32

TABLE I. THE COMPARISON OF TYPICAL TECHNIQUES TO DEFEND AGAINST MEMORY CORRUPTIONS

	No Source Code	No Debug Information	Control Hijacking Attacks	Non-control Data Attacks	Identifying Key Steps
data-flow-integrity [10]	×	–	√	√	×
WIT [8]	×	–	√	√	×
StackGuard [5]	√	√	√	×	×
TaintCheck [13]	√	√	√	×	×
PointerTaint [14]	√	√	√	√	×
CFI [15]	√	√	√	×	×
Clause2007 [12]	√	×	√	√	×
MemCheck [11]	√	×	√	×	×
BinArmor [16]	√	√	√	√	×
PointerScope [1]	√	√	√	×*	√
deExploit	√	√	√	√	√

the program receives the input, it should intuitively parse the input into various fields [21], [22], and take corresponding data structures or variables as references to represent such fields. However, illegal fields in the exploit could overrun their corresponding data structures and overwrite other unintended data structures. That is, illegal fields are not referenced by their corresponding data structures, and the execution of the exploit violates its syntactic format. With this observation, the *deviation between the syntactic format of program input and corresponding data structures* could be a reliable pattern to identify memory corruptions.

The main challenges behind deExploit is to model the deviations between the syntactic format and corresponding data structures. In this paper, we propose to use a fine-grained dynamic tainting technique to model how the exploit is dynamically processed during program execution, dynamically reverse corresponding data structures of program input and then identify overwritten data structures by detecting the *deviation between dynamic processing of exploit and that of benign input*. We implement deExploit in the platform of Bitblaze [23] and perform it to diagnose multiple exploits in the wild. The results show that deExploit works well to diagnose memory corruption exploits.

The novelties and contributions of our paper include three aspects.

- We propose to identify overwritten data structures as a generic approach for memory corruption detection. To make this approach practical at the binary level, we investigated how the syntactic format of program input and its dynamic processing in the execution could be used to identify overwritten data structures, based on the observation that the dynamic execution of memory corruption exploit violates its syntactic format.
- We propose deExploit, a binary level technique for memory corruption exploit diagnosis. deExploit is generic to typical memory corruption attacks, including both control-hijacking and non-control data attacks, and additionally is effective in identifying key attack steps for understanding how the exploit works.
- We implement deExploit with fine-grained dynamic tainting, and apply deExploit to diagnose several exploits in the wild. The experiments show that deExploit is generic to attack types, and effective in identifying key attack steps.

II. PROBLEM SCOPE AND OBSERVATION

A. Problem scope

Let P as an executable, with an unknown memory corruption vulnerability, and exp is the exploit that could trigger the vulnerability and perform malicious attack. deExploit aims to detect the attack in exp, and identify key attack steps for understanding how exp works. For effective exploit diagnosis, deExploit should satisfy the following three requirements.

- **Generic**. Attack detection is a significant step during exploit diagnosis, and deExploit should be generic to attack types, including both control-hijacking and non-control data attacks.
- **Explainable**. Besides attack detection, deExploit aims to be explainable by identifying key attack steps for understanding how the exploit works.
- **Practical**. deExploit should be practical to directly work on binary executables, and does not require the support of source code or debugging information.

B. A motivating example

Figure 1 shows the vulnerable statements in Ghttpd, which is a web server program. In log(), the program declares a 200-byte buffer, named as temp, to represent the request URL. If the request URL contains more than 200 bytes, it could overrun the 200-byte buffer and lead to a memory corruption. Generally, this vulnerability could be exploited by stack smashing to overwrite the return address. In this example, we show a non-control data attack [18] to exploit this vulnerability without control flow diversion.

With the assembly of log(), esi is allocated for ptr that points to the request URL. The serveconnection() is designed to check whether the URL contains substring "/..". If yes, this request is rejected directly. However, the memory corruption occurs after the check, and it indicates a TOCTTOU (Time Of Check To Time Of Use) attack. That is, we can first present a legitimate URL without "/.." to bypass the checking of serveconnection(), and then overwrite esi to point to a malicious URL that could force the program to executable outside the restricted cgi-bin directory.

```
int serveconnection(int sockfd){
   char *ptr; //pointer to the URL.
              //ESI is allocated
              //to this variable
   ...
   if (strstr(ptr,"/..."))
     //reject the request;
   log(...);
   if (strstr(ptr,"cgi-bin"))
     Handle CGI request
}
void log(char *format, ...){
   char temp[200];
   ...
   vsprintf(temp, format, ap);
   // buffer overflow
}
```

```
Assembly of log(…)
push    %ebp
mov     %esp, %ebp
push    %edi
push    %esi
push    %ebx
… stack buffer overflow code
pop     %ebx
pop     %esi //overwritten
pop     %edi
pop     %ebp
ret
```

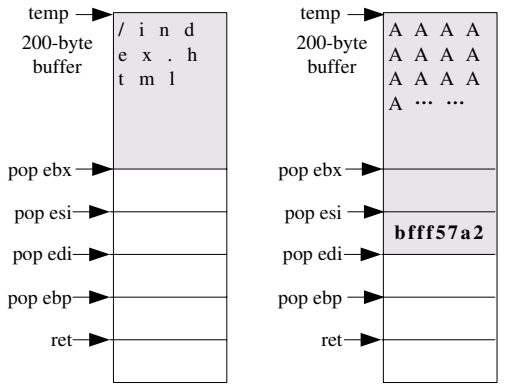
Fig. 1. The vulnerability and non-control data attack on `ghttpd`

The exploit with non-control data attack is represented as `GET AAA...AA\xa2\x57\xff\xbf\x0A\x0A/cgi-bin/../../../../bin/sh`. It contains two parts that are separated with `EOF`. The program converts the first part, `AAA...AA\xa2\x57\xff\xbf`, into a null-terminated string pointed by `ptr` in `serveconnection()`. This substring passes the checking of "/..". When this string is passed to `log()`, it overruns the 200-byte buffer and overwrites the 4-byte memory segment that saves the copy of `esi` to `0xbfff57a2`, which is the address of the second part, `/cgi-bin/../../../../bin/sh`. When `log()` returns, `ptr` points to a `CGI` request, and execute `/bin/sh`.

Many techniques, such as dynamic tainting [13] and control-flow integrity [15], would lose their abilities to detect and diagnose this non-control data attack, because the exploit neither overwrites return address, function pointers, nor violates the control-flow integrity.

C. Our observation

Program inputs usually contain structural information, which is often denoted as *syntactic format* [19], [20]. As syntactic format, the program input consists of several different fields. For example, in the HTTP packet, `GET /index.html`, `GET` refers to the request method, and `/index.html` refers to the request URL. During program executions, these fields should be subsequently referenced by corresponding data structures. In this example, the request URL `/index.html` is referenced by `temp`.



(a) the memory layout of benign input

(b) the memory layout of exploit

Fig. 2. The memory layout of non-control data attack on `ghttpd`

However, for the execution of memory corruption exploit, the value of an illegal field overruns unintended memory segments. That is, the illegal field is not referenced by its corresponding data structure, and some bytes of this field are referenced by unintended data structures. We can observe that the dynamic execution of exploit violates the syntactic format, thus spotting a deviation between the *syntactic format* and *corresponding data structure*.

Take the example in Figure 1 for illustration. We say that the second field in the request refers to the request URL, and the program takes `temp` as reference. During dynamic execution, the memory is always addressed with `base pointer + offsets`, and base pointers point to the memory address of data structures. Figure 2(a) shows the stack layout during the execution of benign input such as `/index.html`, the bytes of request URL is stored in the memory segment of `temp`, and the program would take `temp` as base pointer to address bytes in this request URL.

Figure 2 (b) shows the stack layout during the execution of exploit. The long URL overruns the buffer of `temp`, and overwrites the 4-byte memory segment that saves the copy of `esi` to `0xbffff57a2`. From the syntactic format, such 4-byte `0xbffff57a2` is a portion of the request URL. Meanwhile, the program should take `temp` as base pointer to address these 4 bytes. However, during the dynamic execution of `<pop esi>`, such 4-byte `0xbffff57a2` is addressed with the base pointer specified by `esp`. We can observe that 4-byte from the request URL is referenced by another data structure during the dynamic execution, instead of its corresponding data structure `temp`.

From the example, we observe that the deviation between the *syntactic format* and *corresponding data structure* during dynamic execution seems to be a reliable pattern for exploit diagnosis. However, there are several challenges for our intuitive idea to work at the binary level.

- The basic idea in this paper is to identify overwritten data structures for exploit diagnosis. Without the support of source code or debugging information, data structures are invisible in binary executables. To

effectively define and model the dynamic processing is the first challenge.

- To capture the deviation between the *syntactic format* and *corresponding data structure*, an intuitive idea is to inspect whether input fields are processed by their corresponding data structures during dynamic executions. However, this intuitive idea is limited because we have no idea about how these data structure should be used to represent program input due to complex program logics [22]. Thus, it is another challenge to clearly define the pattern for deviation detection.

III. Design and Implementation

A. Patterns and Deviations

In this section, we propose the design to detect *the deviation between the syntactic format of program input and corresponding data structures* for identifying overwritten data structures.

When a program input is read into the program, such input should be parsed into several independent fields, and each field should be referenced as one unit. To be different with input fields that are specified by the *syntactic format*, we define *dynamic field* to represent the continuous bytes that are always referenced as one unit during execution. The *corresponding data structure* of a *dynamic field* refers to the data structure in the program that is used to reference the *dynamic field* , and the *execution context* refers to instructions as well as call stacks in which the *dynamic field* is referenced.

The dynamic processing to a dynamic field is denoted as an expression with a three-tuple of $\langle f_i, d_j, e_k \rangle$, in which f_i refers to the dynamic field, d_j refers to the corresponding data structure referencing f_i , and e_k refers to execution context. In other words, a three-tuple, $\langle f_i, d_j, e_k \rangle$ indicates that the program takes the data structure d_j to reference the dynamic field f_i in the execution context e_k. With the definitions, for a program `P` with an input `I`, the dynamic processing of `I` by `P` can be modeled as sequences of the three-tuples.

As we demonstrate in the motivation section, the dynamic execution of exploit violates its syntactic format, thus an intuitive idea to identify overwritten data structures is to inspect whether input fields are processed by their corresponding data structures. However, this idea is limited since we do not know what are the corresponding data structures for each fields. In this paper, we design to adopt the execution comparison for deviation detection by comparing the dynamic processing of exploit with that of a benign input.

The dynamic processing of program inputs, to some extent, represents the program logic that is pre-defined. If the execution paths are identical for different benign inputs, the dynamic processing of these two benign inputs should also be identical. Therefore, if we can find out a benign input of which the execution path is identical or at least similar with the execution path of the exploit, we make the dynamic processing of this benign input as a expected dynamic processing patterns, and detect deviations by comparing the dynamic processing of exploit with that of expected benign input. That is, `deExploit` does not directly detect the deviation between corresponding data structures and input fields. As an alternative, `deExploit` first reverses dynamic field and corresponding data structure to model how the program dynamically processing the input, and then detects *deviation between dynamic processing of exploit and that of benign input*. In the following sections, we will present the design and implementation details of `deExploit`.

B. Design Overview

Figure 3 shows the architecture of `deExploit`. In this paper, we assume that the benign input for comparison is available. Actually, how to generate or select inputs for comparison is a challenging problem [24] in software debugging and behavior analysis techniques, and how to obtain the different inputs for comparison is application independent with the scope of this paper.

As shown in Figure 3, firstly we use the fine-grained dynamic tainting to monitor executions. With the fine-grained dynamic tainting, we can capture the propagation of every byte in the input and record the complete execution traces through instrumentation. Second, we analyze the execution context, reverse both the dynamic fields and their corresponding data structures to model the dynamic processing. Third, since the execution contexts are not identical for different executions, we normalize the memory addresses and corresponding data structures for effective comparison. By comparing the dynamic processing of the exploit with that of a benign input, we finally detect overwritten data structure and identify key attack steps.

C. Dynamic tainting and monitoring

In the fine-grained dynamic tainting, we give each input byte a unique taint tag, and record the whole propagation of every byte. Especially, for instructions with multiple source operands, we make the destination operand have the amalgamative taint tags.

During the execution monitoring, we record two types of execution context information, one refers to the run-time call stack and instructions, and the other is the taint records of instruction operands . As the first step in context analysis, we should reconstruct the call stacks. Reconstructing the call stack is a challenge work due to the program logic and obfuscation [25]. For example, sometimes the `call` instruction is used not to invoke a real function, but only as part of a `call/pop`. In this paper, we reconstruct the stack frames by capturing the dynamic balancing of both `ebp` and `esp`. In details, we capture the instructions of `call` as well as `<mov %esp, %ebp>` to identify the allocation of a new stack frame, and we capture the instructions of `<pop %ebp>`(or `leave`) as well as `ret` to identify the exit of a frame.

D. Reversing dynamic fields and corresponding data structures

In recent years, many reverse engineering techniques have been proposed to identify program data structures, and most of them are designed on the intuition that dynamic memory

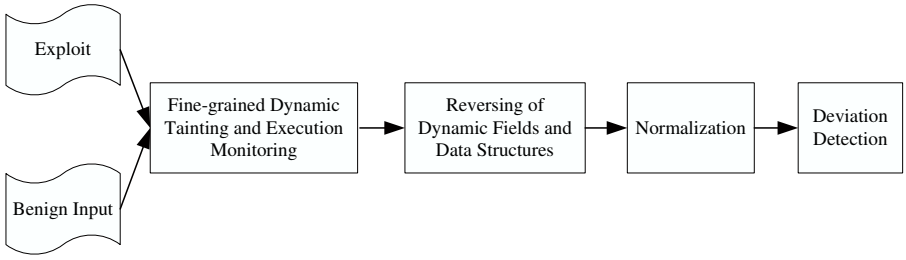


Fig. 3. The overview of deExploit

access patterns reveal the program data structures. For example, Howard [26] identifies base pointers dynamically by tracking the way in which new pointers are derived from existing ones by calculating offset. In this paper, we design to reverse program data structures as well dynamic fields collaboratively, by tracking the fine-grained propagation of program input, and the memory access patterns to such tainted bytes.

Our reversing approach works as follows. For instructions with tainted operands, we first check the tainted byte and identify its memory address as a pointer sink, then identify whether this pointer sink is derived from another pointer by backtracking instructions. If yes, we will make the derived pointer as the pointer sink to further identify its derived pointer, until reaching root pointers that are not derived from another pointers. After identifying base pointers of tainted bytes, we further identify dynamic field with the condition that the offsets of tainted bytes are continuous and their base pointers are identical.

Since the memory segments could be allocated statically (such as global and static variables), or dynamically (such as local variables in the stack and dynamically allocated variables in the heap), the root pointers is identified separately for various data structures as previous techniques [26].

For local variables in the stack, the compiler either uses `ebp` or `esp` to address local variables. Thus, we make `ebp` and `esp` as root pointers for each stack frame. In other words, if a pointer is derived from `ebp` or `esp` by adding an offset, we regard this pointer as the base pointer of a local variable. Please note that address of a memory operand in an x86 instruction is computed as `address = base + (index*scale) + displacement`, where the base and index values are referenced with registers. For special operators such as `lea`, it is not very clear to identify which value refers to the base address. In such situation, we simply select the largest one as the base address as previous techniques [24].

For dynamically located variables in the heap, we monitor the invoking of memory allocations functions, and make the calling instruction as well as return address as allocation site. For example, if an instruction calls `malloc`, we record the memory address of returned value, as well as the instruction that calls `malloc` as the allocation site.

For static variables and the program's global variables, their memories are statically allocated, and typically referenced by instant addresses. However, the memory allocation methods are different for multiple executables. For example, sometimes the memory addresses of statically allocated variables are identical in multiple executions, whereas they could also be relocated for each execution. To ease this problem, we simply take the instruction referencing the statically allocated memory as the alternative representation of statically allocated variables. In other words, if two data structures are both statically allocated and referenced by identical instructions, we would regard these two the same, even the memory addresses of the two statically allocated are not identical.

E. Normalization

The dynamic fields are represented with offset intervals that are specific to certain input, and desirable to normalize structure of dynamic fields.

In previous studies on protocol reversing [21], [22], the tree structure is widely used and well suitable to model input structures. In this paper, we also employ the tree to normalize the structures of specific inputs.

Before being parsed by programs, the tree structure is initialized as one root node. Then the tree is dynamically constructed as the input is dynamically parsed. Whenever a new field is dynamically identified, we search its parent node of which the offset interval is the smallest yet covers the offset interval of the new field, and then insert a new node into the tree as a child node. If no parent is found, we will make the root node as the parent.

In the normalized representation, the dynamic field is represent by the structure of corresponding node in the tree. The structure of a node in the tree structure is denoted as the depth of the node, as well as the ordering number of nodes which have the same parent and are ordered by their offsets.

F. Deviations Detection

As demonstrated in Section III, the dynamic processing of `I` by `P` is modeled as a sequence of the procedure that the program dynamically processes input fields, and each item in this sequence refers to the execution context in which `P` takes the corresponding data structure as the reference to access certain dynamic field.

For two executions, we detect deviations by comparing every pair of three-tuples, until we find that the two items cannot match with each other. After identifying deviations, we believe that some dynamic fields may overwrite

unintended data structures, thus memory corruptions have occurred. We will regard the dynamic fields and their corresponding data structures, as well as the execution contexts, as the key attack steps.

IV. Evaluation

We implement deExploit on the platform of Bitblaze, and evaluate deExploit using several real-world vulnerable programs. Table II shows the vulnerable programs, the CVE-ID of vulnerabilities, and the attack techniques of the exploits. To evaluate that deExploit is able to be generic to attack types, and be explainable to identifying significant attack steps, we select exploits with both control-hijacking and non-control data attacks, and advanced exploits that consist of multiple attack steps, such as ROP attack.

A. Experiment results

The exploits and experiment results are summarized in Tables III. The running time includes two items, one is the time to capture execution traces by dynamic tainting, and the other is the time to detect overwritten data structures. From Table III, the performance of dynamic tainting is overhead and the sizes of some traces are very large, especially for programs on the Windows platform. For example, it causes more than 20 minutes to capture the trace for coolplayer, and the trace file contains more than 1G bytes.

The numbers of deviations between the execution of exploit and that of benign input are shown in Table III. For example, for the exploit of CVE-2008-3408 with ROP attack, deExploit identifies 16 deviations by comparing the dynamic processing of exploit with that of benign input. Among the 16 deviations, we successfully identify the key attack steps, such as constructing the parameters of SetProcessDEPPolicy. In the following section, we will discuss the details.

For the exploit of CVE-2006-3747, we encounter a false negative and miss the key attack step that overwriting the return address. The reason is that the overwritten value is the memory address of tainted bytes, instead of the tainted bytes themselves. In other words, the overwritten return address is control-depended on the program input, instead of data-depended on the program input. This is a representative under-tainting problem.

B. Case studies

In this section, we take several case studies to illustrate that 1)deExploit is generic to attack steps, especially non-control data attack that is hard to detect by previous techniques, and 2) deExploit is able to identify key attack steps for understanding how the attack works, such as ROP attack.

1) Non-control data attacks on gzip and ghttpd.: There is a buffer overflow between two global variables in the gzip program. The execution of exploit could make gzip fail and output an unexpect result, which is a non-control DoS attack.

The program input is a string that refers to the filename. In general, this string consists of one field, and it should be accessed as one unit during dynamic executions. For benign input, we find that gzip successively takes two different data structures to reference this string, and the base pointers are 0x807f680 and 0x807faa0, respectively. We could observe that the dynamic field is consistent with that as syntactic format.

```
b7e8d250   mov    0x4(%esp),%ecx
b7e8d254   mov    %ecx,%eax
b7e8d283   mov    (%eax),%ecx
           M@0x0807faa0[0x6966676e]   T1[1056, 1057, 1058, 1059]
```

Fig. 4. The execution context of the non-control DoS attack on gzip

```
804a497    pop    %ebx
           M@0xbffe360c[0x61616161]   T1[322, 323,324,325]
804a498    pop    %esi
           M@0xbffe3610[0xbfff57a2]   T1[326, 327, 328, 329]
804a499    pop    %edi    M@0xbffe3614[0xbfb559c7]   T0
804a49a    pop    %ebp    M@0xbffe3618[0xbffe77a8]   T0
804a49b    ret            M@0xbffe361c[0x08049ee9]   T0
```

Fig. 5. The execution context of the non-control data attack on ghttpd

Let us move to the execution of exploit. Figure 4 shows the trace segment. Before illustrating the experiment result, let us explain the format of execution trace first. In Figure 4, each line presents the memory address of instruction, the instruction, and the operands. Take the instruction <mov (%eax), %ecx> for illustration, the memory address of this instruction is 0xb7e8d283. In this instruction, there is one source operand, which is M@0x807faa0[69666763]. M means the operand is read from the memory, and the value is 0x69966763 with the address of 0x807faa0. Following the operand is the taint information. T1 means that the operand is tainted and propagated from the program input. The taint tags for each byte are followed and separated by a comma. In our approach, we make the offset from the beginning as the taint tag for each input byte. In Figure 4, the taint tags for the four bytes in the source operand are 1056, 1057, 1058, 1059, respectively. It indicates that these four bytes are propagated from the input bytes of which the offsets range from 1056 to 1059. The formats of following execution trace segments are identical as shown in Figure 4.

As shown in Figure 4, the memory address of the tainted 4-byte is 0x807faa0, and is derived from eax. Backtracking the data propagation, we find that eax is copied from the memory segment 0x4(%esp), and this memory segment is specified with esp. As esp refers to a root pointer, the memory segment 0x4(%esp) indicated a base pointer. That is, the base pointer of such tainted 4-byte is stored in the memory segment 0x4(%esp). We say that the program takes a data structure of which the base address is 0x807faa0 to reference the tainted 4-byte. Similarly, we finally find that the portion of which the offset starts from 1056 is dynamically referenced by the base pointer 0x807faa0, and a deviation is detected. We believe that the data structure of which the base address is 0x807faa0 is likely overwritten.

TABLE II. VULNERABLE PROGRAMS AND EXPLOITS

Programs	CVE ID	Attack Techniques	Platform
Apache-1.3.31	CVE- 2004-0940	Stack overflow	Linux
3CTfpdSvc-0.11	N/A	Stack overflow	Windows
knet-1.04b	CVE- 2005-0575	SEH exploit; ROP attack	Windows
ghttpd-1.4.3	CVE- 2001-0820	Non-control data attack	Linux
gzip-1.2.4	CVE- 2001-1228	Non-control DoS attack	Linux
coolplayer-2.18	CVE- 2008-3408	Stack overflow; ROP attack	Windows
Apache-1.3.31	CVE-2006-3747	Off-by-one DoS attack	Windows

TABLE III. SUMMARY OF EVALUATION

Vulnerabilities	Exploits & benign inputs	Running Time	Trace Size	Total Instructions	Deviations
CVE-2004-0940	Exploit	2m10s; 5m10s	1,327,642,497	9,135,126	2
	Benign input	1m36s, 4m16s	943,327,312	8,772,515	
3CTfpdSvc	Exploit	5m16s, 1m30s	60,404	427,540	1
	Benign input	6m32s, 1m50s	70,009	645,617	
CVE-2005-0575	Exploit	22m40s, 7m30s	3,248,019,963	30,995,325	6
	Benign input	18m16s, 6m16s	2,346,406,479	20,370,905	
CVE-2001-0820	Exploit	1m16s, 30s	57,688	424,945	5
	Benign input	1m10s, 20s	55,928	411,353	
CVE-2001-1228	Exploit	1m16s, 26s	42,830	308,773	1
	Benign input	58s, 12s	23,830	169,371	
CVE-2008-3408	Exploit	20m50s, 6m30s	1,091,527	7,909,697	16
	Benign input	17m30s, 6m10s	621,939	4,959,121	
CVE-2006-3747	Exploit	1m52s, 4m30s	871,912,814	6,297, 809	Missed
	Benign input	18m16s, 6m16s	2,346,406,479	20,370,905	

Let us show another example that is shown in the motivation section. We construct the exploit with non-control data attack as described in [18], and the trace segment is shown in Figure 5.

As shown in Figure 5, two fields (their offset intervals are [322, 325] and [326, 329], respectively) are dynamically identified, because their base pointers are derived from `esp` and the memory addresses are `0xbffe360c` and `0xbffe3610`, respectively. Moreover, we find that these two dynamic fields are never present in benign executions. Thus, we believe that two overwritten data structures have been detected.

Please note that these two non-control data attacks are hard to detect by previous techniques. For example, in `gzip` program, the overflow just occurs between two global variables and leads to no control flow violation. Pointer tainting [14] can detect some non-control data attacks by detecting tainted pointers, but it cannot detect the exploit of `gzip` because the pointer is taintless. PointerScope [1] aims to diagnose exploits by identifying the conflicts of pointer types. However, in this example, both the base pointers, `0x807f680` and `0x807faa0`, are correctly used and there is no conflict.

2) ROP attacks on `coolplayer`: As shown in Table III, for the ROP attack on `CVE-2008-3408`, `deExploit` identifies 16 deviations by comparing the dynamic processing of exploit with that of benign input. In this section, we will illustrate these deviations could be used to help understand how the exploit works.

Figure 6 shows the trace segment of `coolplayer`, and ten data structures are dynamically identified. For example, for the instruction `ret`, of which the instruction address is `0x4089b6`, the operand is addressed by the root pointer `esp`, thus the operand of is dynamically identified as a dynamic field. With taint tags, we further find that the dynamic field contains four bytes, and the offsets range from 260 to 263. For the instruction `<pop %ebx>`, of which the instruction address is `0x77f167a6`, the operand is addressed by `esp`, thus the operand is dynamically identified as a data structure, and the base pointer points to `0x001321f0`. The dynamic field of this data structure consists of four bytes, and the offsets range from 264 to 267. Similarly, we can identify other eight data structures.

Figure 6 shows the execution context of these ten overwritten data structures. The virtual lines refer to the control transfer, and the solid lines refer to the data propagation. By examining the context, we find that the `eip` that points to `SetProcessDEPPolicy` is propagated from the input with offsets ranging from 276 to 279. The argument of `SetProcessDEPPolicy`, specified with `<mov 0x8(%ebp), %eax>`, are propagated from the input and the offsets range from 264 to 267. Capturing these overwritten data structures, we identify that the invoking of `SetProcessDEPPolicy` is constructed by attacks.

C. False positive and false negative

1) False positive: In our experiments, we could detect extra overwritten data structures that seem to bring false positives to attack detection and attack steps identification. Actually, these overwritten data structures should not be regarded as false alarms, because the overwriting really happens except that the references of overwritten data structures belong to the normal program logics instead of attacks. For example, in the experiment of `coolplayer`, we found six overwritten data structures that are not attack steps.

Figure 7 shows two of the six overwritten data structures. From the execution traces shown in Figure 7, we observe that after the execution of memory corruption, the program references two data structures to call another function of

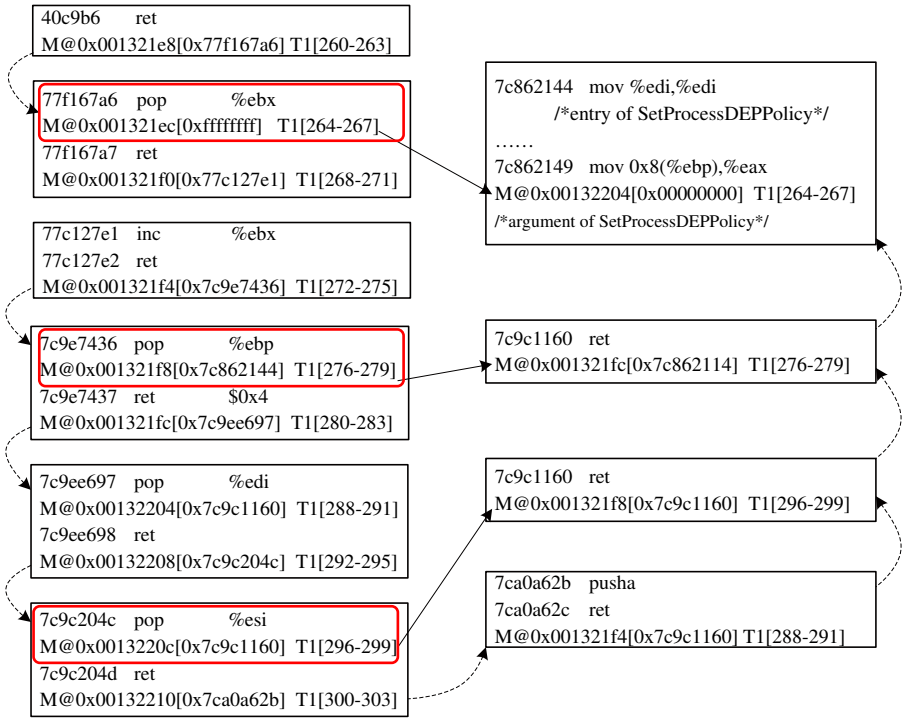


Fig. 6. The execution context of ROP attack on `coolplayer`

```
40c990    mov    0x11c(%esp),%ecx
          M@0x001321f4[0x7c9e7436]  T1[272, 273, 274, 275]
40c997    lea    0xc(%esp),%edx
40c99b    push   %ecx
40c99c    mov    0x118(%esp),%eax
          M@0x001321f4[0x7c9e7436]  T1[264, 265, 266, 267]
40c9a3    push   %edx
40c9a4    push   %eax
40c9a5    call   0x0040bf70
```

Fig. 7. Detected overwritten data structures that are not attack key steps

```
2699      static char *escape_absolute_uri
          (ap_pool *p, char *uri, unsigned scheme)
2700      {
          ……
2737      token[0] = cp = ap_pstrdup(p, cp);
2738      while (*cp && c < 5) {
2739            if (*cp == '?') {
2740            token[++c] = cp + 1;
/* cp points to the user input, and could overwrite the
return address due to the off-by-one vulnerability */
2741            *cp = '\0';
          ……
2744      }
```

Fig. 8. The source code of vulnerability `CVE-2006-3747`

which the entrance address is `0x40bf70`. With the impact of stack smashing, the arguments assigned to the function (`0x40bf70`) are overwritten. These two data structures are really overwritten, but they may not the attack steps.

2) False negative: In our experiments, we encounter one example of false negative. To illustrate the detail of `CVE-2006-3747`, we present the source code of the vulnerability in Figure 8.

This vulnerability is an off-by-one memory corruption. In the code segment, `cp` is defined as a pointer that points to the URL reading from the program input, `token` is defined as an array with 5 elements. The buggy operation `++c` in line 2740 could lead to an out-of-bounds memory operation, because the index of `token` could be 5 if `c` equals to 4. With certain compilers, this out-of-bounds memory operation could be exploited for control-hijacking. For example, by examining the binary of `Apache-1.3.31` on the Windows platform, the memory segment of `token` is just followed by the return address. That is, the off-by-one element is just the return address, and out-of-bounds memory operation could overwrite to hijack the execution.

This is a representative under-tainting problem [27] caused by control flow [28], and we cannot capture the memory corruptions that are not data-depended on tainted bytes. To solve this problem, we could improve the tainting propagation by enabling the control dependencies [29].

V. Limitations and Discussions

In this paper, deExploit is able to identify most of memory corruptions caused by overwriting. However, there are other memory corruptions caused by over-reading. For example, the famous vulnerability, heart-bleed (CVE-2014-0160) in OpenSSL is caused by a memory disclosure attack and does not overwrite any data. Recent advanced attack, such as JIT-ROP [30], also involves memory disclosure attack. For such exploits, we cannot identify the steps of memory disclosure.

The observation behind execution comparison is that the dynamic processing of program input should be identical within the specific execution contexts and paths. However, it may not be sustainable for special cases and how to construct benign input for comparison become a challenging work. For example, many images and audio consists of record sequences [31]. In such cases, the execution loop is very complex and deExploit cannot recognize the dependencies and many deviations could arise .

VI. Related work

The defence against memory corruption has always been a research focus in the past decades. In this section, we mainly discuss the most related work with deExploit.

A. Reverse engineering

Researchers have proposed many data structure reversing techniques, such as ASI [32], TIE [33], REWARDS [34], Howard [26], and so on. With static analysis, ASI attempts to pinpoint memory segments depending on access patterns. As memory addresses are referenced, ASI identifies an approximate mapping of variable-like locations. In addition, ASI can further identify types with the type information from system calls and well-known library functions such as libc. Similar with the idea behind ASI, REWARDS [34] propagates type information from known parameter types. Howard [26] constructs symbol tables by capturing access patterns of different data structures.

B. Memory corruption detection and diagnosis

Many researchers from different areas have proposed many kinds of protection techniques, including language security [35], library safety [36], data flow integrity [10], writing-integrity [8] and so on. Most of these protection techniques require the support of source code [8], [10], and cannot be employed on binary executables.

At the binary level, control-flow-integrity [15] and dynamic taint analysis [37] are two representative techniques to detect attacks and diagnose vulnerabilities. For example, TaintCheck [13] can detect the attacks that try to overwrite the return address, function pointers, and format string vulnerabilities. Control-flow-integrity is effective to detect control-hijacking attacks with low overhead [38]. However, existing techniques may not work well in defensing advanced exploits such as non-control data attacks [18]. Researchers propose pointer tainting [14] to detect the non-control data attacks, but pointer tainting still brings large false negatives and false positives, because some bytes of the input used as a pointer are not rare [39].

Besides attack detection, techniques are proposed for understanding exploit. For example, Argos [17] tracks the propagation of unsafe data to identify the invalid use and further generate accurate detection signatures for the exploits that are immune to payload mutations. Prospector [40] identifies the bytes contributing to an overflow, which is very useful for exploit diagnosis and similar with deExploit. However, the detections of memory corruption behind Argos and Prospector is still based on control flow diversion.

Memcheck [11] detects the overlap between the source and destination memory regions, but fails to detect non-control memory corruptions among local variables. Clause et al. propose the memory protection techniques through tainting memory allocation [12] instead of tainting program input. Without the symbol table, this approach may not detect the non-control memory corruptions on the stack.

BinArmor [16] is a binary-level memory protection technique through reversing program data structures and protect these data structures with write-integrity like WIT [8]. BinArmor defends against both control-hijacking and non-control data attacks, and is practical to work at the binary level. However, BinArmor detect memory corruptions at the point of vulnerability executon, and may fail to identify key attack steps. For example, BinArmor cannot identify the overwritten data structure in SEH exception handler, and it cannot identify all overwritten return addresses in the ROP attack.

Execution comparison is widely used for software debugging, security analysis, which are close related to our paper. Differential slicing isolates the causal path of state differences leading to the observed difference [24]. Dual slicing compares two executions and extracts the differing dependencies between them, which have been used to debug concurrency bugs [41]. The main challenge to existing execution comparison is that the number of different program states could be very large. In this paper, we do not select the detailed program states for comparison. Instead, we model the program execution of how it parses program inputs. Within the application of memory corruption diagnosis, our technique could propose more precise information to assist analysts to understand the attack steps and diagnose exploits.

VII. Conclusion

Exploit diagnosis is a significant yet time-consuming effort during software security defence. To automate this progress, exploit diagnosis should be generic to attack steps, and explainable to identify key attack steps for advanced exploits. In this paper, we investigate how the syntactic format and dynamic processing of program input could be used to identify overwritten data structures, and propose deExploit, a binary-level exploit diagnosis approach. Evaluations on several realistic exploits in the wild show that deExploit could be generic to attack types and effectively identify key attack steps.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (nos. 61303213, 61373169), and the foundation of Key Laboratory of Information Assurance Technology (KJ-13-104).

REFERENCES

[1] M. Zhang, A. Prakash, X. Li, Z. Liang, and H. Yin, "Identifying and Analyzing Pointer Misuses for Sophisticated Memory-corruption Exploit Diagnosis," in *the 19th Annual Network & Distributed System Security Symposium*, 2012.

[2] A. Prakash, H. Yin, and Z. Liang, "Enforcing system-wide control flow integrity for exploit detection and diagnosis," in *the 8th ACM SIGSAC symposium on Information, computer and communications security*. New York, New York, USA: ACM Press, 2013, pp. 311–322.

[3] D. Brumley, J. Newsome, and D. Song, "Towards automatic generation of vulnerability-based signatures," in *2006 IEEE Symposium on Security and Privacy*. IEEE, 2006, pp. 15–30.

[4] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: securing software by blocking bad input," in *the 21st symposium on Operating systems principles*, 2007, pp. 117–130.

[5] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security Symposium*, 1998.

[6] S. Bhatkar and D. DuVarney, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *USENIX Security Symposium*, 2003.

[7] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy*, 2014.

[8] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing Memory Error Exploits with WIT," in *2008 IEEE Symposium on Security and Privacy*, 2008, pp. 263–277.

[9] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *USENIX Security Symposium*, 2009, pp. 51–66.

[10] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *OSDI*, 2006, pp. 147–160.

[11] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *USENIX Annual Technical Conference*, 2005, pp. 17–30.

[12] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective memory protection using dynamic tainting," in *ASE*, 2007, pp. 284–292.

[13] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *NDSS*, 2005.

[14] N. Nakka, Z. Kalbarczyk, and R. Iyer, "Defeating Memory Corruption Attacks via Pointer Taintedness Detection," in *DSN*, 2005, pp. 378–387.

[15] M. Abadi, M. Budiu, . Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security*, vol. 13, no. 1, pp. 1–40, 2009.

[16] A. Slowinska, T. Stancescu, and H. Bos, "Body armor for binaries: preventing buffer overflows without recompilation," in *USENIX Annual Technical Conference*, 2012.

[17] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation," in *the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006, pp. 15–27.

[18] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-hijacking attacks are realistic threats," in *USENIX Security Symposium*, 2005.

[19] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI*, vol. 43, 2008, pp. 206–215.

[20] Z. Lin and X. Zhang, "Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications," *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 688–703, 2010.

[21] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *CCS*, 2007, pp. 317–329.

[22] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *NDSS*, 2008.

[23] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *the 4th International Conference on Information Systems Security*, 2008.

[24] N. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, "Differential Slicing: Identifying Causal Execution Differences for Security Applications," in *2011 IEEE Symposium on Security and Privacy*, 2011, pp. 347–362.

[25] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," *the 23rd USENIX Security Symposium*, 2014.

[26] A. Slowinska, T. Stancescu, and H. Bos, "Howard: a dynamic excavator for reverse engineering data structures," in *the 2011 Network and Distributed System Security Symposium*, 2011.

[27] E. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.

[28] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: Dynamic taint analysis with targeted control-flow propagation." in *NDSS*, 2011.

[29] I. Haller, A. Slowinska, H. Bos, and M. M. Neugschwandtner, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violation," in *the 22nd USENIX Security Symposium*, 2013, pp. 49–64.

[30] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 574–588.

[31] W. Cui, M. Peinado, K. Chen, H. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *CCS*, 2008, pp. 391–402.

[32] G. Ramalingam, J. Field, and F. Tip, "Aggregate structure identification and its application to program analysis," in *the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 119–132.

[33] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *the 2011 Network and Distributed System Security Symposium*, 2011.

[34] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *the 17th Network and Distributed System Security Symposium*, 2010.

[35] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *USENIX Annual Technical Conference*, 2002, pp. 275–288.

[36] T. Tsai and N. Singh, "Libsafe: transparent system-wide protection against buffer overflow attacks," in *DSN*, 2002, pp. 541–550.

[37] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *the 11th international conference on Architectural support for programming languages and operating systems*, 2004, pp. 85–96.

[38] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 559–573.

[39] A. Slowinska and H. Bos, "Pointless tainting? Evaluating the practicality of pointer tainting," in *EuroSys*, 2009, pp. 61–74.

[40] A. Slowinska and H. Bos, "The Age of Data: pinpointing guilty bytes in polymorphic buffer overflows on heap or stack," in *Annual Computer Security Applications Conference*, 2007, pp. 487–500.

[41] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan, "Analyzing concurrency bugs using dual slicing," in *the 19th international symposium on Software testing and analysis*, 2010, pp. 253–264.