The Journal of Systems and Software 124 (2017) 153–168

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

deExploit: Identifying misuses of input data to diagnose memory-corruption exploits at the binary level☆

Run Wang[a,b,c], Pei Liu[d], Lei Zhao[a,b,c,*], Yueqiang Cheng[e], Lina Wang[a,b,c]

[a] Stake Key Laboratory of Software Engineering, Wuhan University, 430072, China
[b] Key Laboratory of Aerospace Information Security and Trust Computing, Wuhan University, 430072, China
[c] School of Computer, Wuhan University, 430072, China
[d] Changjiang River Scientific Research Institute, Wuhan, 430010, China
[e] APL Software, CA, 94043, USA

ARTICLE INFO

Article history:
Received 31 December 2015
Revised 6 November 2016
Accepted 13 November 2016
Available online 14 November 2016

Keywords:
Software vulnerability
Exploit diagnosis
Memory corruption
Reverse engineering

ABSTRACT

Memory-corruption exploits are one of the major threats to the Internet security. Once an exploit has been detected, exploit diagnosis techniques can be used to identify the unknown vulnerability and attack vector. In the security landscape, exploit diagnosis is always performed by third-party security experts who cannot access the source code. This makes binary-level exploit diagnosis a time-consuming and error-prone process. Despite considerable efforts to defend against exploits, automatic exploit diagnosis remains a significant challenge. In this paper, we propose a novel insight for detecting memory corruption at the binary level by identifying the misuses of input data and present an exploit diagnosis approach called deExploit. Our approach requires no knowledge of the source code or debugging information. For exploit diagnosis, deExploit is generic in terms of the detection of both control-flow-hijack and data-oriented exploits. In addition, deExploit automatically provides precise information regarding the corruption point, the memory operation that causes the corruption, and the key attack steps used to bypass existing defense mechanisms. We implement deExploit and perform it to diagnose multiple realistic exploits. The results show that deExploit is able to diagnose memory-corruption exploits.

1. Introduction

A vulnerability is a type of software bug that can be manipulated by attackers to alter the intended software behavior in a malicious way (Brumley et al., 2006). An exploit is an actual input that triggers a vulnerability, typically with malicious intent and devastating consequences, such as starting a shell or escalating privileges. Among the causes of software vulnerabilities, memory corruption is the most significant reason (Szekeres et al., 2013), and CERT advisories demonstrated that memory-corruption exploits constitute approximately half of all reported attacks (Veen et al., 2012).

Once an exploit for an unknown vulnerability has been detected, security experts generally try to diagnose the exploit by replaying the attack. This enables them to detect the exploit, identify the root cause of the vulnerability, and finally provide a patch or security advisory (Xu et al., 2005; Sezer et al., 2007). Exploit diagnosis is a significant problem in the security landscape. For example, almost all mainstream operating systems and applications have adopted patching to remove newly discovered vulnerabilities. Several intrusion-detection techniques (Costa et al., 2007) and input filters (Brumley et al., 2006) have been developed to provide temporary protections after an exploit has been identified, but before patches have been properly applied. All these defense mechanisms require exploit diagnosis to provide specific information about the vulnerability and the exploit.

This paper focuses on a binary-level approach for exploit diagnosis. We require exploit diagnosis to work at the binary level because source-code-based approaches are impractical in the security landscape. In recent years, many automatic debugging techniques have been proposed to assist in fixing vulnerabilities and generating software patches (Wu et al., 2014; Cui et al., 2016; Jun et al., 2016). For example, RETracer (Cui et al., 2016) reconstructs program semantics from core dumps and examines how the program input contributes to program crashes. CREDAL (Jun et al., 2016) analyzes the core dump of a crashed program without the assumption of memory integrity. However, the distribution and applica-

☆ The preliminary version of this paper is published in the 39th IEEE Computer Society International Conference on Computers, Software & Applications (COMPSAC 2015).

* Corresponding author.
E-mail address: leizhao@whu.edu.cn (L. Zhao).

http://dx.doi.org/10.1016/j.jss.2016.11.026
0164-1212/

tion of patching are very slow. Gkantsidis et al. (2006) showed that it takes about 24 h for 80% of the unique observed IPs to notice when a new Windows Update patch is available for download, whereas current exploits such as the Slammer worm compromise almost all vulnerable hosts within minutes (Moore et al., 2003). For emergency response, many other defense techniques are required for temporary protection. These defenses are always provided by security experts from third-party organizations who cannot access the source code (Brumley et al., 2006). Compared with debugging techniques that require the support of source code, binary-level exploit diagnosis is preferable because it enables security experts who cannot access the source code to automatically analyze the exploits and vulnerabilities. Additionally, these debugging techniques are unsatisfactory for special security applications such as identifying the attack vector employed by the exploit to generate exploit signatures.

Exploit diagnosis is time-consuming, tedious, and error-prone (Zhang et al., 2012; Prakash et al., 2013). In particular, binary-level exploit diagnosis remains a challenging and difficult problem that should be automated wherever possible. Understanding the complexities of a vulnerability has consistently proven to be very complicated for humans at the source-code level (Jun et al., 2016), let alone the binary level. In recent decades, various defense techniques have been proposed for memory-corruption exploits. However, most previous techniques are insufficient for automatic exploit diagnosis because of the following challenges.

- Practicality for binary programs. Enforcing memory safety is a generic approach that can stop all types of memory corruption (Szekeres et al., 2013). Representative techniques include pointer bounds (Cowan et al., 2003) and object bounds (Akritidis et al., 2009). Approximation techniques, which are weaker than memory safety, have also been proposed, such as WIT (Akritidis et al., 2008) and DFI (Castro et al., 2006). However, these techniques require the support of source code and are impractical for binary programs.
- Genericity to attack types. At the binary level, most attack detection techniques are designed by capturing the attack manifestations (Haller et al., 2013). For example, TaintCheck (Newsome and Song, 2005) can detect attacks that try to overwrite return addresses or function pointers. CFI (Abadi et al., 2009) detects violations of control-flow-integrity to mitigate control-flow-hijack attacks, and many practical CFI techniques (Mohan et al., 2015; van der Veen et al., 2015; Zhang et al., 2013; Payer et al., 2015; Tice et al., 2014) have been proposed. Traditionally, memory-corruption vulnerabilities are exploited for control-flow-hijack attacks (Szekeres et al., 2013). These vulnerabilities can also be exploited for data-oriented attacks (also known as non-control data attacks Chen et al., 2005). Because the attack manifestation of data-oriented attacks is difficult to detect, many detection techniques may lose their capabilities (Slowinska et al., 2012).
- Provision of precise information about the vulnerability and exploit. Recent studies have developed the binary-level memory safety or weaker policies in which the program data structures are reversed (e.g., BinArmor (Slowinska et al., 2012), StackArmor (Chen et al., 2015), and Data-delineation (Gopan et al., 2015)). These techniques can detect both control-flow-hijack and data-oriented attacks. However, the problem scope of exploit diagnosis is different from that of exploit detection. Most detection techniques can detect attacks whenever the vulnerabilities are triggered, but fail to provide precise information about the exploited vulnerability. Moreover, security mechanisms such as stack protection (Cowan et al., 1998), address-space-layout randomization (ASLR) (Bhatkar et al., 2003), and data execution protection (DEP) (Data Execution Protection, 2013) lead to increasingly sophisticated exploits (Buchanan et al., 2008; Snow et al., 2013; Bittau et al., 2014). These sophisticated exploits always consist of multiple attack steps to circumvent existing defense mechanisms (Zhang et al., 2012). Information about the vulnerability and exploit are important for exploit diagnosis in terms of localizing the root causes of vulnerabilities and generating exploit signatures; however, providing such information has rarely been considered.

This paper presents a binary-level exploit diagnosis approach called deExploit. This approach is generic in terms of the detection of various attacks, including both control-flow-hijack and data-oriented exploits. In addition to detection, deExploit can also localize the root cause of an unknown vulnerability by providing information regarding the corruption point in the instructions. deExploit can identify the key attack steps, allowing us to understand the attack vector employed by the exploit to circumvent existing defense mechanisms.

The insight behind deExploit is that ***the exploitation for a memory-corruption vulnerability often involves misusing program inputs as the value of undefined data structures and the misuse of input data can be employed to detect memory corruption and diagnose exploits***. More specifically, program inputs are usually organized as the syntax format definition, and these inputs consist of several independent fields. When a program receives its input, it should subsequently reference various data structures to store and manipulate these fields (Caballero et al., 2007). However, the attacker always maliciously constructs the exploit to corrupt and control sensitive data (such as the return address and function pointers), with the purpose of bypassing system protections and executing unintended instructions. That is, certain data in the exploit are misused as the value of other data structures. With this observation, modeling how the program manipulates the input and identifying the misuse of input data is an attractive approach for detecting memory-corruption attacks and diagnosing exploits at the binary level. In this study, we employ a fine-grained dynamic tainting approach to dynamically reverse the data structures. For each input field, we construct a set of data structures that are referenced by the program to access this field. For the execution of the exploit, once a data structure reference to a field is not defined in benign executions, we detect an invalid reference and finally identify the misuse of input data for exploit detection and diagnosis.

This paper is an extended version of our previously published paper (Zhao et al., 2015). In our previous work, we proposed the detection of exploits and the identification of key attack steps. For effective exploit diagnosis, we are also interested in how the misuse of input data could be used to localize the root causes of software vulnerabilities. Meanwhile, our previous approach causes many false alarms because not every data misuse is part of a key attack step. These two problems have motivated us to investigate the localization of the root causes of vulnerabilities and the reduction of false alarms when identifying key attack steps.

The novelty and contributions of our paper include three aspects.

- This paper presents deExploit, a binary-level exploit diagnosis approach. This approach is generic in terms of the detection of various attacks, including both control-flow-hijack and data-oriented exploits. Moreover, deExploit can localize the root cause of an unknown vulnerability by providing information regarding the corruption point, vulnerable data structures, and suspicious memory operations. It can also identify the key attack steps to help understand the attack vector employed by the exploit to circumvent existing defense mechanisms.
- To detect memory corruption at the binary level, we propose the novel insight of identifying the misuse of input data. The

Table 1
Comparison of typical techniques to defend against memory corruption.

		No source code	No debugging info	Control-hijacking attacks	Data-oriented attacks	Localizing root causes	Identifying key steps
Program protection	DFI (Castro et al., 2006)	×	–	√	√	√	×
	WIT (Akritidis et al., 2008)	×	–	√	√	√	×
	BinArmor (Slowinska et al., 2012)	√	√	√	√	√	×
	StackArmor (Chen et al., 2015)	√	√	√	√	√	×
	Delineation (Gopan et al., 2015)	√	√	√	√	√	×
	Clause2007 (Clause et al., 2007)	√	×	√	√	×	×
Attack detection	TaintCheck (Newsome and Song, 2005)	√	√	√	×	×	×
	PointerTaint (Nakka et al., 2005)	√	√	√	√	×	×
	CFI (Abadi et al., 2009)	√	√	√	×	×	×
	binCFI (Zhang and Sekar, 2013)	√	√	√	×	×	×
Memory debugging	Memcheck (Seward and Nethercote, 2005)	√	×	√	×	×	×
	Memsherlock (Sezer et al., 2007)	×	×	√	√	√	×
Exploit diagnosis	PointerScope (Zhang et al., 2012)	√	√	√	×	×	√
	deExploit	√	√	√	√	√	√

insight is that the exploitation for a memory-corruption vulnerability often involves misusing input data as the value of undefined data structures and the misuse of input data can be employed to detect memory corruption and diagnose exploits. Based on this observation, we investigate how the dynamic reverse engineering of program data structures can be used to model the method whereby the program manipulates the exploit. We then detect memory-corruption attacks by identifying the invalid data structure referenced by the exploit.

- We implement deExploit and utilize it to diagnose several realistic exploits. The results show that deExploit can detect both control-flow-hijack and data-oriented exploits and that it is applicable in localizing the root causes and identifying the key attack steps.

2. Motivation and observations

2.1. Background

In the arms race between offense and defense over the last 30 years, exploitation techniques for vulnerabilities have been constantly evolving, and a set of defenses has been developed to fight against vulnerabilities and exploits (see Table 1).

As the root cause of memory-corruption vulnerabilities is the lack of memory safety in unsafe programming languages (Szekeres et al., 2013), eliminating memory errors and enforcing memory safety are generic approaches, which can detect and prevent all types of memory-corruption attacks. Typically, these generic techniques are based on precise objects and pointer-to analysis, which require the support of source code. Taking advantage of reverse engineering techniques, researchers have proposed several binary-level approaches that enable memory safety protection without the support of source code. These binary-level techniques (such as StackArmor (Chen et al., 2015) and Data-delineation (Gopan et al., 2015)) first recover and identify the program data structures, and then enforce runtime memory safety via static analysis of the recovered objects.

Software protection techniques aim to prevent memory corruption. Another important type of defense mechanism is to detect unknown exploits or memory-corruption attacks. Among these detection techniques, most of them are based on the signatures of attack manifestations (Slowinska et al., 2012). For example, TaintCheck (Newsome and Song, 2005) detects attacks by capturing the behavior of overwriting return addresses or function pointers, and CFI (Abadi et al., 2009) detects attacks by identifying the violation of control flow integrity.

This paper focuses on exploit diagnosis. The problem scope of exploit diagnosis is different from those of software protection and exploit detection. Protection and detection techniques can prevent and detect unknown exploits, whereas exploit diagnosis identifies the unknown vulnerability and determines the attack vector upon the detection of a new exploit. Therefore, an exploit diagnosis technique is required to detect the exploit, as well as to localize the root cause of vulnerabilities and identify the key attack steps regarding how the exploits circumvent existing defense mechanisms. Compared with attack detection, the localization of vulnerability root causes and the identification of key attack steps have rarely been studied. In summary, exploit diagnosis typically involves at least three stages: detecting the exploit, localizing the root causes of the unknown vulnerability, and identifying the key attack steps to help understand the attack vector.

For the problem of exploit detection, few previous binary-level detection techniques can be generic to the detection of various attacks. In addition to the control-flow-hijack attacks that are often considered in previous techniques, data-oriented attacks (Hu et al., 2015; 2016) (also known as non-control data attacks Chen et al., 2005) are another type of memory-corruption attack. Recent studies have shown that data-oriented exploits can be automatically generated for a given memory-corruption vulnerability (Hu et al., 2015; 2016; Carlini et al., 2015; Evans et al., 2015). Moreover, recent exploits have utilized data-oriented attacks as their key steps (Subverting without EIP, 2014; PanguTeam, 2015). For example, the control-flow bending (CFB) (Carlini et al., 2015) and Control-Jujutsu (Evans et al., 2015) attacks can leverage memory-corruption vulnerabilities to construct an arbitrary code execution even when fine-grained CFI is enforced. A recent exploit on Internet Explorer (IE) 10 changes a single byte to enable arbitrary code execution (Subverting without EIP, 2014). An iOS jailbreaking exploit also involves a data-oriented attack to bypass code-signing validation (PanguTeam, 2015). Compared with control-flow-hijack exploits, the attack manifestations of data-oriented exploits are concealed and difficult to detect. Most previous detection techniques (such as dynamic tainting (Newsome and Song, 2005) and CFI (Abadi et al., 2009)) are unable to capture the attack manifestations of data-oriented attacks. In summary, an effective exploit diagnosis should be generic in terms of attacks types, including both control-hijacking and data-oriented exploits.

For automatic vulnerability analysis, researchers have proposed many automatic debugging techniques for the localization of memory errors. Memsherlock (Sezer et al., 2007) can automatically identify unknown memory-corruption vulnerabilities upon the detection of an exploit. This identifies the corruption point and describes how the malicious input exploits the vulnerability, which

requires the support of source code. Considering security applications (such as the generation of vulnerability signatures), the source code and debugging symbols are rarely available. Unlike program developers, who can debug and fix software bugs with the support of source code, security experts always perform the exploit diagnosis on binary programs directly. Thus, the above source-code-based techniques may be impractical for programs without available source code, such as commercial-off-the-shelf (COTS) binaries or legacy code. CBones (Kil et al., 2007) extracts and verifies program structural constraints to detect security bugs. It is based on the attack manifestation in which the exploit violates the structural constraints. PointerScope (Zhang et al., 2012) aims to identify key attack steps by detecting pointer misuses. These two techniques are practical for binaries. However, CBones and PointerScope may fail in some data-oriented attacks. For example, if memory corruption occurs between two stack objects during a data-oriented attack, no violation will be detected by CBones or PointerScope.

2.2. Problem scope

In the exploit diagnosis problem, we are given a vulnerable binary program *P* and an exploit *exp* for an unknown vulnerability. For the program *P*, neither the source code nor debugging information is available. The goal of exploit diagnosis is to assist security experts (especially experts from third-party organizations, for whom the source code is not available) by automatically providing precise information regarding the corruption point in the instructions, the memory operation that causes the corruption, and the key attack steps used to bypass existing defense mechanisms.

For improved effectiveness demonstrated in Section 2.1, we have designed deExploit to satisfy the following three requirements.

- **Practicality**. To enable security experts who cannot access the source code to perform automatic analysis of exploits and vulnerabilities, we designed deExploit to be practical at the binary level.
- **Genericity**. To overcome the problem of exploit detection, we designed deExploit to be generic in terms of the detection of various attacks, including both control-flow-hijack and data-oriented exploits.
- **Explanatory ability**. For effective diagnosis of vulnerabilities and exploits, we designed deExploit to provide explanatory information by identifying the corruption point in the instructions and the memory operation that caused the corruption. Additionally, deExploit can also identify the key attack steps, enabling us to understand the attack vector employed by the exploit.

2.3. Motivating example

Fig. 1 shows the vulnerable statements in *ghttpd*, which is a web server program. In the function *serveconnection*, the program defines a pointer *ptr* and dereferences it to access the value of the request URL. In the function *log*, the program converts the received URL (pointed to by *ptr*) into a formatted string, and saves it in a 200-byte buffer called *temp* for system logging. If the URL contains more than 200 bytes, it will overrun the buffer and corrupt adjacent memory regions. Typically, this vulnerability could be exploited by overwriting the return address and hijacking the control flow. Besides control-flow-hijack attacks, this vulnerability can also be exploited for data-oriented attacks.

In the program, the function *serveconnection* is defined to check whether the URL contains the substring"/..". If so, this request URL is directly rejected to prevent the execution of programs that are

```
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
}
```

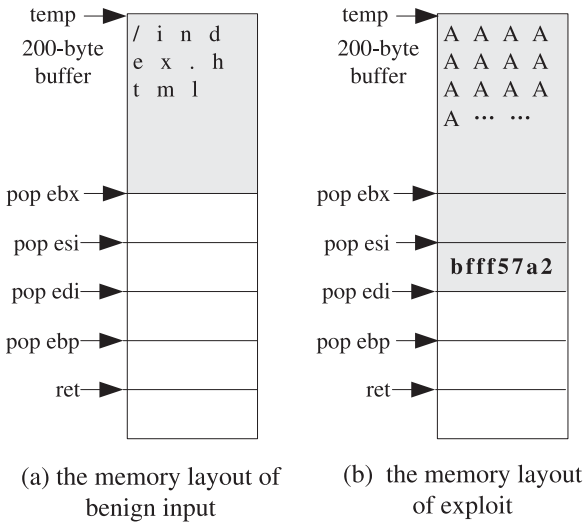
```
Assembly of log(…)
push    %ebp
mov     %esp, %ebp
push    %edi
push    %esi
push    %ebx
… stack buffer overflow code
pop     %ebx
pop     %esi //overwritten
pop     %edi
pop     %ebp
ret
```

Fig. 1. Data-oriented attack on *ghttpd*.

outside the restricted *cgi-bin* directory. However, the memory corruption that occurs after the security check indicates a Time-of-Check-To-Time-Of-Use (TOCTTOU) attack. In the TOCTTOU attack, an attacker first presents a legitimate URL without"/.." to bypass the *serveconnection* check procedure. The attacker then overwrites the saved *ptr* and makes it point to a malicious URL containing"/..". Through this way, the attacker can successfully force the program to execute outside the restricted *cgi-bin* directory, without any CFI violations.

As described in the original data-oriented exploit (Chen et al., 2005), the program assigned *esi* to hold the copy of *ptr* and pushed *esi* into the stack when *serveconnection* called *log*. A long URL then overwrites the saved *esi* and makes *ptr* point to an unintended string. However, the original exploit (Chen et al., 2005) no longer works on current compiled programs, because *esi* no longer saves the copy of *ptr*. For ease of presentation, we consider the original exploit to illustrate our motivation and observation. In Section 5, we will present the working exploit for evaluation.

The data-oriented exploit is represented as *GET AAA..AA\x29\xd7\xff\xbf\x20\x20/cgi-bin/../../../../bin/sh*. The substring following *GET* contains two parts split by a blank space (*\x20*). The program converts the first part, *AAA..AA\x29\xd7\xff\xbf*, into a null-terminated string and stores it in the memory region pointed to by *ptr* (as shown in *serveconnection*). This string passes the checking of"/..". In *log*, the long URL overruns the 200-byte buffer and overwrites the 4-byte memory region of the saved *esi* to *0xbfffd729*. When *log* returns, the value of *ptr* is overwritten as *0xbfffd729*, which is the memory address of the second part



(a) the memory layout of benign input

(b) the memory layout of exploit

Fig. 2. Memory layout of the data-oriented exploit.

/cgi-bin/../../../../bin/sh. Later, the program will execute the program */cgi-bin/../../../../bin/sh* specified by *ptr* and start a shell.

In diagnosing this data-oriented exploit, few previous techniques have effectively identified the corruption point and attack vector that allows the execution of */cgi-bin/../../../../bin/sh* to pass the security check. For example, most binary-level detection techniques (such as dynamic tainting (Suh et al., 2004; Newsome and Song, 2005) and CFI (Abadi et al., 2009)) are unable to detect and diagnose this data-oriented exploit, because the exploit does not perform return-address overwriting, function-pointer overwriting, or CFI violations. System protection (such as the stack cookie) can detect and prevent this attack, but cannot provide precise information regarding the corruption point and attack vector. Binary-level techniques such as StackArmor (Chen et al., 2015) and Data-delineation (Gopan et al., 2015) can detect the corruption point, but fail to determine the attack vector.

2.4. Our observation

Program inputs usually contain structural information, and they consist of several independent fields. This structural information is often denoted as *syntax format*.

Definition 1. The syntax format refers to the structure of program inputs consisting of multiple fields. Each field is semantically independent. Typically, input fields are split by separators or based on a specific length.

When a program receives its input, it should subsequently parse the input into various fields, and allocate different memory regions and corresponding data structures to store these fields (Lin et al., 2008). At the binary level, the program will dereference corresponding pointers to access these fields.

For illustration, let us consider the motivating example. According to the syntax format of the HTTP request, in the first line of the HTTP packet, e.g., *GET /index.html HTTP/1.1*, */index.html* refers to the request URL, and *HTTP/1.1* refers to the HTTP version. These two fields are split by a blank space. After reading the input, the program uses corresponding data structures to store */index.html* and *HTTP/1.1*. The memory layout is shown in Fig. 2(a). The program allocates a 200-byte buffer to store */index.htm*, and it dereferences the pointer of *temp* to access the URL during the execution.

Similarly, for the exploit represented by *GET AAA...AA\x29\xd7\xff\xbf\x20\x20/cgi-bin/../../../../bin/sh*, the first part *AAA...AA\x29\xd7\xff\xbf* refers to the request URL, and the second part */cgi-bin/../../../../bin/sh* should be the version of HTTP. As defined by the program semantics, the program will convert and save the request URL *(AAA...AA\x29\xd7\xff\xbf)* into the memory region at *temp*. However, because the request URL exceeds 200 bytes, the execution of the exploit will lead to a buffer overflow on *temp*. Fig. 2(b) shows the stack layout. The long URL overruns the buffer *temp* and overwrites the saved *esi* to *0xbffffd729*. Later, the instruction ⟨*pop esi*⟩ accesses this 4-byte *0xbffffd729* through the pointer specified by *esp*. That is, the instruction interprets the 4-byte *0xbffffd729* as the value of *esi*, which holds the copy of *ptr*. However, in the syntax format definition, the 4-byte *0xbffffd729* should be a portion of the request URL. As the request URL is stored in *temp*, the program should dereference the pointer of *temp* to access these four bytes. It is observed that the violation where the 4-byte *0xbffffd729* is misused and interpreted as the value of *ptr*. Following this observation, we further find that the misuse of input data is a generic feature for memory corruption, because the attacker always seeks to corrupt and control some sensitive program data with user inputs for malicious intent. Therefore, identifying the misuse of input data is an attractive approach for detecting and diagnosing exploits. However, several challenges must be overcome for our intuitive idea to work at the binary level.

- The first step to detect the misuse of input data is to construct a set of data structures for each input field. The difficulty is that we do not know what the data structures are and what the mapping relationship (between the data structures and input fields) should be. Thus, the first challenge is to define the detection policy clearly.
- There is no high-level abstraction such as data structures or types in binary programs. Thus, the second challenge is to make our idea practical at the binary level.
- For effective exploit diagnosis, it is still unknown how misused data could be used to localize vulnerability root causes and identify key attack steps.

3. Overview of deExploit

3.1. Detection policy

The detection policy is based on the property that data structures referenced by the program to access input fields are defined as program semantics. For an input field *f*, the set of data structures that are referenced by the program to store and manipulate *f* is called the data-structure-reference set for *f*, denoted as *DS(f)*. Because *DS(f)* is predefined by the program semantics, whenever *f* is accessed, the data structure referenced by the program to access *f* should be in *DS(f)*. Otherwise, we can detect invalid data structure references.

The main challenge for implementing the detection policy is to determine the data-structure-reference set for each input field, which is difficult without a thorough understanding of program semantics. Without the support of high-level abstractions, this work is more challenging for binary programs. In addition, input fields in a benign input are commonly parsed and manipulated into multiple data structures, because of the complex program logic. That is, an independent field is commonly interpreted as multiple units during program execution. This phenomenon exacerbates the difficulties for our detection policy. For this problem, we define a dynamic field to model how the program manipulates fields dynamically.

Definition 2. A dynamic field refers to the input portions that are always manipulated as one unit during the program execution.

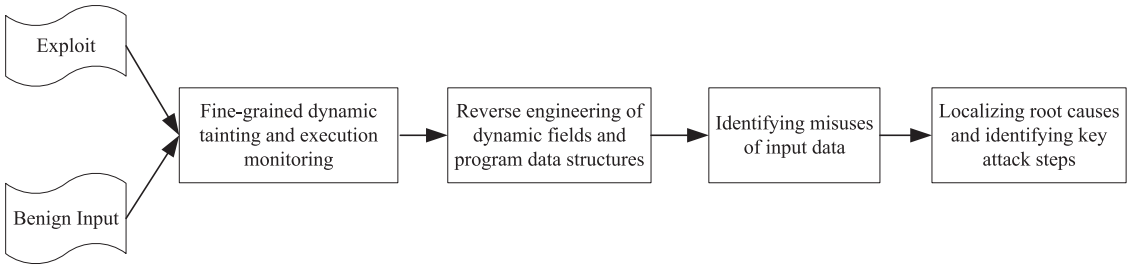


Fig. 3. Overview of deExploit.

Note that the definition of a dynamic field is different from that of an input field in the syntax format. According to Definition 1, an input field is defined as per the syntax format definition. According to Definition 2, a dynamic field is specified by the way the program manipulates the input. Moreover, an input field defined by the syntax format may be manipulated as multiple dynamic fields during the execution.

When the program receives the input, it should subsequently parse the input into multiple dynamic fields step by step. Specifically, the program first regards the input as one entire dynamic field, and then parses the whole dynamic field into multiple dynamic fields according to the definition of program semantics. We define a tuple of $\langle f, d, e\rangle$ to model the way in which ***the program references data structures to access a dynamic field***. In this tuple, f, d, and e refer to the dynamic field, the data structure used to store and manipulate f, and the execution context, respectively. A tuple $\langle f, d, e\rangle$ indicates that the program references data structure d to access dynamic field f within execution context e. At the binary level, the data structure is always represented by corresponding pointers as in previous techniques. The tuple $\langle f, d, e\rangle$ also indicates that the program dereferences the pointer of d to access the dynamic field f within the execution context e. To model the progressive method whereby a program P parses and manipulates an input I into multiple dynamic fields step by step, we model the dynamic execution of I by P as sequences of $\langle f, d, e\rangle$.

To detect invalid data structure references, we employ an execution comparison and construct the data-structure-reference set for each dynamic field. Our observation is that the way the program manipulates the benign input is just the criterion for violation detection. The data structure references for each input field are predefined by the program semantics. If the program parses and manipulates the two inputs in the same way, the data structure references to the same field should also be identical. Therefore, we can construct the data-structure-reference set for each input field from benign executions, and then detect the violation of invalid references to the exploit through execution comparison.

3.2. Approach overview

Fig. 3 shows an overview of deExploit.

As shown in Fig. 3, deExploit first generates or selects a benign input. The execution of this benign input is used to detect the misuse of exploit data. Second, deExploit employs a fine-grained dynamic tainting technique to monitor executions. The fine-grained dynamic tainting enables us to capture the propagation of every byte in the input and record the complete execution trace. Third, by analyzing the memory access patterns, deExploit dynamically reverses dynamic fields, as well as memory regions and corresponding data structures used to store these dynamic fields. Then, deExploit constructs the data-structure-reference set for each dynamic field from the benign execution, and inspects the dynamic execution of the exploit to identify the invalid references. Finally, deExploit localizes the root causes of vulnerabilities and identifies the key attack steps by examining the dependency relationship between misused input data.

4. Design and implementations

4.1. Generation of benign executions

Program inputs often consist of multiple optional fields as per the syntax format definition (Cui et al., 2008). Moreover, the value of some input fields can determine the program behavior (Carbin and Rinard, 2010). For effective execution comparison, we should generate or select a benign execution in which the program logic is identical to that of the exploit.

The technique of generating or selecting executions for comparison is another significant problem in many research areas such as software debugging. Many techniques have been proposed for overcoming this problem. For example, a benign input could be calculated with symbolic execution by collecting execution traces and execution constraints (Godefroid et al., 2008). If the test suite is available, a much simpler approach is selecting the benign execution by comparing the execution traces (Sumner et al., 2011). In this study, we generate benign inputs by mutating the exploit. Specifically, we use a parser to obtain the syntax format of the exploit. Then, we mutate the value for each field to generate inputs that cause no crash. Simultaneously, we compare the execution traces of these benign inputs and select the one with the execution trace most similar to that of the exploit.

4.2. Dynamic tainting and monitoring

As a detection policy, deExploit is designed to identify invalid data structure references for the detection of memory-corruption attacks. The challenge is that there are no data structures in the binary programs and we cannot determine how the dynamic fields are accessed. In deExploit, we employ a fine-grained dynamic tainting and monitoring technique to dynamically reverse the dynamic fields as well their corresponding data structures.

Traditional dynamic tainting uses a taint tag to indicate whether the operand is derived from the program input (Suh et al., 2004; Newsome and Song, 2005). However, this is unsatisfactory for dynamic reverse engineering. The fine-grained dynamic tainting in deExploit gives each input byte a unique taint tag and records the whole propagation of each byte. In particular, for instructions with multiple source operands, deExploit taints the destination operand with the combined taint tags.

During the execution monitoring, deExploit records two types of information. One is the executed instructions, their operands, and the memory addresses of instructions. The other is the taint records of the operands.

4.3. Reversing dynamic fields and program data structures

In recent years, many reverse engineering techniques have been proposed to reverse the syntax format of the program input and

program data structures. Most of these techniques are designed on the intuition that the way the program manipulates the input reveals a wealth of information about the syntax format (Caballero et al., 2007). For example, Howard (Slowinska et al., 2011) dynamically identifies base pointers by tracking the way in which new pointers are derived from existing ones by calculating the offset. REWARDS (Lin et al., 2010) dynamically identifies input fields based on the observation that various input fields are typically handled in different execution contexts.

As in previous techniques (Slowinska et al., 2011), we denote the pointer that points to the base of a memory object as the base pointer of the corresponding data structure. Based on the observation that the program will dereference the same base pointer to access the consecutive bytes in one dynamic field, we propose to reverse the program data structures as well as the dynamic fields collaboratively. Specifically, deExploit first determines the tainted operand in the instructions and its memory address. For example, the addressing mode of an x86 instruction is computed as *address = base + (index*scale) + displacement*, where the base and index values are indicated by registers. deExploit checks whether this address is derived from another address by backtracking the calculation on memory addresses. If so, deExploit further checks whether the address is derived from a third one, until deExploit reaches the base pointer. After identifying the base pointers of tainted bytes, deExploit regards the consecutive bytes addressed with the same base pointer as a dynamic field.

However, identifying the base pointer for a data structure is a challenging task, because the contexts where types of memory objects are allocated are significantly different. For overcoming this problem, we must recognize the execution contexts coupled with instructions to recover the base pointers for data structures (Slowinska et al., 2011).

Memory objects can be allocated either statically (such as global and static variables) or dynamically (such as local variables on the stack and dynamic variables on the heap). Static memory objects are not reused, so they can be uniquely identified by their address. However, the stack and heap are constantly reused, with the allocation and deallocation of stack frames and heap objects. Based on previous studies (Slowinska et al., 2011), the insight for recognizing reallocation is that the memory-allocation contexts are different for types of memory objects; thus, the data structures on the stack and heap should be coupled with specific execution contexts.

The program typically allocates a new frame for every invocation of a function, and the frame holds all local variables on the stack. In this study, deExploit captures the dynamic balancing of both *ebp* and *esp* to reconstruct the stack frame. Specifically, deExploit captures the call instruction as well as $< mov\,\%esp, \%ebp >$ to identify the allocation of a new stack frame, and the $\langle pop\,\%ebp \rangle$ (or *leave*) instruction as well as *ret* to determine the deallocation of a frame. For local variables on the stack, the compiler usually uses either *ebp* or *esp* to specify their base pointers. Thus, if a pointer is derived from *ebp* or *esp* by adding an offset, i.e., 0x4(*%ebp*), deExploit will regard this pointer as a base pointer to a local variable.

For variables that are allocated dynamically on the heap, deExploit monitors the invocation of memory-allocation functions to capture the allocation of objects. The returned value of a memory-allocation function refers to the address of the allocated object. Then, deExploit integrates the returned address and the invocation of memory-allocation functions to represent the variable. For example, if an instruction calls *malloc*, the returned value refers to the memory address of the allocated object. Following the instruction that calls *malloc* and analyzing the assignment of the returned value, deExploit can further identify the base pointer for the allocated object.

4.4. Detecting the misuse of input data

Through dynamic reversing, deExploit can identify the dynamic fields indicated by the offset intervals of the program input. As the length of the input field is always variable, we cannot directly compare dynamic fields for violation detection. For this problem, we employ a tree structure, as in previous studies on protocol reversing (Lin et al., 2008), to normalize the structure of dynamic fields. This is because the program subsequently parses the input into several fields step by step, as in the construction of the tree structure.

In the normalization step, deExploit regards the entire input as one field and initializes the tree structure as a single root node. Once a new field has been identified, deExploit search for the parent node with the smallest offset interval that can cover the offset interval of the new field. Then, deExploit inserts a new node into the tree as a child node. If no parent node is found, deExploit will treat the root node as the parent. In the tree, deExploit queues the nodes in ascending order of their offsets to the dynamic fields. After the normalization, the dynamic field is represented by the depth and ordering number of the node in the whole tree.

With the 3-tuple $\langle f, d, e \rangle$, deExploit can detect two types of violations. The first is where a dynamic field identified in the execution of the exploit is undefined in benign executions. The second is where a data structure referenced by the program to a certain dynamic field is not present in benign executions. The first violation is typically caused by the references to overwritten data structures. That is, if a dynamic field is undefined, it indicates that this dynamic field is likely to be a portion of another dynamic field. When the program references the corrupted data structure, it will interpret this portion as an independent yet undefined dynamic field. The second violation indicates an invalid pointer dereference. The invalid pointer can be triggered by exploiting a vulnerability such as in the exploitation for a use-after-free vulnerability, and the invalid pointer can also be generated by another memory corruption such as overwriting a pointer.

In the motivating example, deExploit detects two violations. One is caused by the 4-byte *0xbffffd729*, because these four bytes should be a portion of the request URL. As the 4-byte *0xbffffd729* belongs to the request URL, the program should dereference the pointer of *temp* to access *0xbffffd729*; actually, the program accesses *0xbffffd729* by dereferencing *esp* and interpreted *0xbffffd729* as the value of another data structure. The other violation is the access to */cgi-bin/../../../../bin/sh*, because dereferencing *ptr* to access */cgi-bin/../../../../bin/sh* is not defined in the benign execution.

4.5. Localizing root causes and identifying key attack steps

4.5.1. Localizing root causes

To localize the root causes of vulnerabilities, we design deExploit to provide information regarding the corruption point in the instructions (i.e., the memory operation that causes corruption), the data dependencies of how the exploit reaches the point, and the context of the conditions that should be satisfied to trigger the vulnerability. In a general exploitation model, the first step in exploiting a memory-corruption vulnerability is to make a pointer invalid. Therefore, the basic approach to localize root causes of vulnerabilities is to identify the generation and dereference of an invalid pointer (Yong and Horwitz, 2003).

The misuse of input data is caused by invalid pointer dereferences. To identify invalid pointers, we need to analyze the cause-effect relationship between the invalid pointer and the data misuse for various types of memory-corruption vulnerabilities.

For spatial memory errors, the invalid pointer dereferences lead to corruption on other data structures. That is, dereferencing the pointer to access the misused data is not the root cause; rather,

Table 2
Vulnerable programs and exploits.

Vulnerable programs	CVE-ID	Upgraded exploit	Exploitation attack	Platform
Apache-1.3.31	CVE-2004-0940	Oct 2004	Stack overflow	Ubuntu 10.04
3CTfpdSvc-0.11	N/A	Mar 2012	Stack overflow	Windows XP
knet-1.04b	CVE-2005-0575	Apr 2013	SEH exploit; ROP attack	Windows XP
ghttpd-1.4.3	CVE-2001-0820	Aug 2015	Non-control data attack	Ubuntu 10.04
gzip-1.2.4	CVE-2001-1228	Nov 2001	Non-control DoS attack	Ubuntu 10.04
coolplayer-2.18	CVE- 2008-3408	Jan 2011	Stack overflow; ROP attack	Windows XP
Apache-1.3.31	CVE-2006-3747	Jul 2006	Off-by-one DoS attack	Windows XP
PuTTY 0.65	CVE-2016-2563	Jun 2016	Stack overflow	Windows XP
ProFTPD 1.3.3a	CVE-2010-4221	Dec 2010	Int-to-buffer overflow	CentOS 6.8

the root cause is the invalid pointer dereference that causes memory corruption. To localize the root causes of the spatial memory errors, we enabled deExploit to highlight the data dependency relationship for the misused data and identified the propagation of suspicious data.

For temporal memory errors, attackers often exploit the vulnerability by forcing a pointer to a deallocated or reallocated memory region. The invalid pointer-dereference is just the root cause. From the perspective of debugging, temporal memory errors are easily captured by analyzing the allocation and deallocation sequences in the execution of an exploit. Therefore, this paper focuses on the diagnosis of memory corruption exploits caused by spatial memory errors.

Our approach can detect multiple misuses of data. Some of them are caused by direct memory corruption, and others may subsequently be affected by the execution of corrupted data. For example, the misuse of */cgi-bin/../../../../bin/sh* is affected by the previous corruption that overwrites the pointer to */cgi-bin/../../../../bin/sh*. Because triggering an invalid pointer is always the first step of the exploitation, the analysis of the first detected misuse is more useful for the localization of root causes.

In particular, deExploit can go deeper to identify more valuable information such as the probable size of the buffer. For misused data, deExploit first determines the undefined dynamic field, and then regards its parent node as the input field to which it belongs. By analyzing the root pointers of the dynamic field and belonging field, deExploit can further infer the size of the vulnerable data structure.

4.5.2. Identifying key attack steps

Based on the observation that attack steps are closely related to each other and are generally integrated into a complete attack vector, deExploit is designed to identify the key attack steps by capturing the dependency relationships between misused data. It captures two main types of dependencies: control dependencies and data dependencies.

Control dependencies include the control transfer, function pointers such as the return address, calling for API functions, and system calls. Data dependencies include both direct data dependencies and pointer propagations. It is common for tainted values to be used as pointers to address other memory segments. Thus, deExploit captures both direct data and indirect data propagation between misused data to construct the whole attack vector.

For each set of misused data, deExploit first identifies the dynamic field, the dereferenced pointer that accesses this field, and the instruction that takes this field as operands. Then, deExploit analyzes the data dependencies and determines whether the dynamic field or its dereferenced pointer is data-dependent on another dynamic field. deExploit also analyzes the control dependencies and determines whether the instruction address is data-dependent on another dynamic field.

In the motivating example, deExploit detects two misuses of data, the 4-byte *0xbffffd729*, and */cgi-bin/../../../../bin/sh*. By analyzing the dependency relationship, deExploit further identifies that the address of the pointer to */cgi-bin/../../../../bin/sh* is specified only by *0xbffffd729*. This indicates that *0xbffffd729* is used as a springboard to execute */cgi-bin/../../../../bin/sh*.

5. Evaluation

We evaluated the effectiveness of deExploit with several binary programs. In addition, we considered different types of vulnerabilities and different exploitation techniques.

Table 2 lists the vulnerable programs, the *CVE-ID* of vulnerabilities, the update dates of exploits, and the exploitation techniques. Note that exploitation techniques are constantly evolving with the enhancement of system protection, and exploits are constantly updated to bypass these protections. For example, the original exploit for the vulnerability *CVE-2001-0120* no longer works, and a recent study (Hu et al., 2015) updated a working exploit. In the evaluation, we selected only the latest and working exploits from previous literature and exploit database systems (Offensive security exploit database archive, 2016). Table 2 lists the update dates of these exploits.

We implemented deExploit on the platform of BitBlaze (Song et al., 2008), which consists of two components: a dynamic taint analysis component (called TEMU) and a static analysis component (called Vine). We implemented the dynamic monitoring in deExploit as a plugin of BitBlaze and designed a fine-grained dynamic tainting approach. With the dynamic monitoring, we obtained execution traces including instructions, operands, and tainting information. To detect the misuse of input data, we implemented another component in deExploit as an off-line analysis tool based on execution traces.

5.1. Experimental results

Table 3 summarizes the experimental results. The runtime includes two items: the time required to capture execution traces by dynamic tainting and the time required to detect misused data. From Table 3, it is apparent that the performance of dynamic tainting had a large weight, and some traces were large, especially for Windows programs. For example, it took more than 20 min to capture the trace for *coolplayer*, and the trace file contained more than 1 GB of data.

Table 3 also presents the number of misused input data. Let us take the exploit for vulnerability *CVE-2008-3408* for illustration. It is observed that deExploit identified 16 misuses of input data. With the first detected misuse, deExploit successfully localized the corruption point in the instructions and the suspicious memory operation causing memory corruption. In all 16 misuses, deExploit successfully identified the key attack steps such as the construction of parameters for the invocation of the function *SetProcessDEPPolicy*. Further details are presented in the case study in Section 5.2.

Table 3
Summary of evaluation.

Vulnerabilities	Exploits & benign inputs	Running time	Trace size	Total instructions	Misuses of input data
CVE-2004-0940	Exploit	2 min 10 s, 5 min 10 s	1,327,642,497	9,135,126	2
	Benign input	1 min 36 s, 4 min 16 s	943,327,312	8,772,515	
3CTfpdSvc	Exploit	5 min 16 s, 1 min 30 s	60,404	427,540	2
	Benign input	6 min 32 s, 1 min 50 s	70,009	645,617	
CVE-2005-0575	Exploit	22 min 40 s, 7 min 30 s	3,248,019,963	30,995,325	6
	Benign input	18 min 16 s, 6 min 16 s	2,346,406,479	20,370,905	
CVE-2001-0820	Exploit	1 min 16 s, 30 s	57,688	424,945	8
	Benign input	1 min 10 s, 20 s	55,928	411,353	
CVE-2001-1228	Exploit	1 min 16s, 26s	42,830	308,773	1
	Benign input	58 s, 12 s	23,830	169,371	
CVE-2008-3408	Exploit	20 min 50 s, 6 min 30 s	1,091,527	7,909,697	16
	Benign input	17 min30 s, 6 min10 s	621,939	4,959,121	
CVE-2006-3747	Exploit	4 min 30 s,1 min 52 s	871,912,814	6,297, 809	Missed
	Benign input	18 min 16 s, 6 min 16 s	2,346,406,479	20,370,905	
CVE-2016-2563	Exploit	6 min 50 s, 2 min 16s	125,773	597, 389	4
	Benign input	6 min 16 s, 1 min 55 s	120,169	564,073	
CVE-2010-4221	Exploit	1 min 27 s, 45 s	82,752	566,314	14
	Benign input	1 min 16 s, 45 s	78,346	529,836	

As shown in Table 3, deExploit encountered a false negative for the exploit for vulnerability *CVE-2006-3747*. This false negative indicates that deExploit failed to detect memory corruption through identifying misuses of input data. To examine the reasons, we find that the overwritten return address is control-dependent, rather than data-dependent, on the program input. That is, the under-tainting problem causes this negative. We present more details and discussions in Section 5.3.

The vulnerability *CVE-2010-4221* is caused by another type of memory corruption called integer-overflow-to-buffer-overflow (Zhang et al., 2010). Specifically, one of the root causes of vulnerability *CVE-2010-4221* is an integer overflow in which an unsafe calculation from a signed integer to an unsigned integer can mistakenly convert a negative number to a large positive number. When the vulnerable program receives a long buffer and copies it to the memory region, it leads to a buffer overflow and corrupts sensitive data such as the saved *ebp* and return address. In this example, deExploit identified 14 misused dynamic fields, including the overwritten return address, the overwritten saved *ebp*, and the misused dynamic fields for constructing an ROP attack. However, deExploit cannot provide information showing that an exploit should trigger the integer vulnerability in advance before triggering this vulnerability.

5.2. Case studies

In this section, we present two case studies to demonstrate that deExploit can detect both control-flow-hijack and data-oriented exploits, especially the data-oriented exploit, which is not easy to detect with previous techniques. Additionally, we present a third case study to demonstrate that deExploit can localize the root causes and identify the key attack steps, which enables us to understand the working procedures of attacks such as ROP.

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

Fig. 5. Execution traces for the data-oriented attack on *gzip*.

5.2.1. Data-oriented attack on gzip

The *gzip* program contains a buffer overflow between two global variables. We constructed an exploit with a long input filename of 2000 bytes, which is a data-oriented attack leading to a denial-of-service (DoS).

As the syntax format, the program input consists of one field indicating the filename. For a benign input, *gzip* subsequently takes two data structures to store this string, and the base addresses of the two data structures are *0x807f680* and *0x807faa0*. Note that the input field defined as the syntax format is manipulated as a dynamic field.

Next, we discuss the execution of the exploit. Fig. 5 shows the trace segment. Before illustrating the experimental results, we explain the format of the execution trace. In Fig. 5, each line represents the memory address of the instruction and operands. As an illustration, consider the instruction $< mov(\%eax), \%ecx >$, whose memory address is *0xb7e8d283*. *M@0x807faa0[69666763]* refers to the source operand, where *M* indicates that the operand is read from memory, and the value is *0x69966763* with address

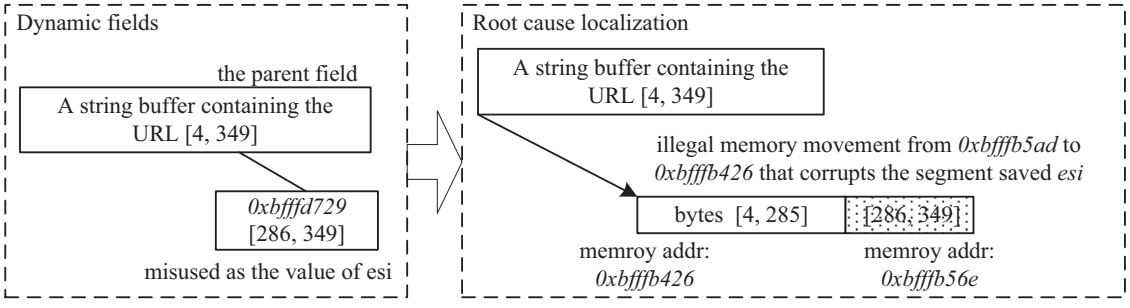


Fig. 4. Localization of the vulnerability root causes for the *ghttpd* exploit.

0x807faa0. The following is the taint information. *T1* means that the operand is tainted and propagated from the program input. Taint tags for each byte are followed and separated by a comma. In this study, deExploit sets the offset from the beginning as the taint tag for each input byte. That is, for an exploit containing 2000 bytes, the offset for each byte ranges from 0 to 1999. In Fig. 5, taint tags for the four bytes of the source operand are 1056, 1057, 1058, and 1059, respectively. This indicates that these four bytes are propagated from the input bytes with offsets from 1056-1059.

As shown in Fig. 5, the memory address of the tainted 4-byte is *0x807faa0*, and is derived from *eax*. Backtracking the data propagation, deExploit finds that the program moves the value of another memory segment to *eax*, and the address of this memory segment is specified by *0x4(%esp)*. As *esp* holding the stack pointer, *0x4(%esp)* will be the base pointer. Then, deExploit identifies the offsets of this dynamic field addressed by *0x4(%esp)* ranging from 1056 to 1999. Because this dynamic field (with the offset interval [1056, 1999]) is not defined in benign executions, a violation is detected. This violation indicates that the exploit has overwritten the data structure of which the base address is *0x807faa0*.

According to the detection policy, it is observed that this violation is of the first type. The input portion [1056, 1999] is likely to corrupt another data structure, and this input portion should be a part of the filename. Through analyzing the execution traces, deExploit identifies that the memory movement from *0x807f680* to *0x807faa0* is the latest write to the memory region at *0x807faa0*. Then deExploit can infer that this memory movement leads to a memory overlap and this is likely to be the cause of such memory corruption. Additionally, deExploit can infer that the size of the vulnerable buffer will probably be 1056 by calculating the offsets between the two memory regions.

Please note that the buffer size is 1024. That is, the input portion [1024, 1055] should have overwritten additional data structures. However, the program did not access the memory regions ranging from *0x807f680* to *0x807faa0*; therefore, deExploit could not capture these misuses of input data.

5.2.2. Data-oriented attack on ghttpd

As mentioned in Section 2, the original exploit no longer works on our platform. In the evaluation, we present a working data-oriented exploit that was demonstrated in a recent study (Hu et al., 2015).

The original exploit leverages a TOCTTOU attack to bypass the security checking by corrupting the pointer that points to the URL. In the original exploit (Chen et al., 2005), the vulnerable program takes *esi* as the pointer to the URL. Thus, an attacker can corrupt and control the pointer by overwriting the saved *esi*. Unfortunately, on newer platforms (such as Ubuntu 10.04), the compiler does not assign *esi* to save the pointer according to the assembly code. Instead, the compiler allocates a local variable at *−0x4160(%ebp)* to save the pointer. This change indicates another data-oriented attack. As shown in Fig. 6, an exploit corrupts *ebp* to pivot the stack frame and forces the pointer addressed by *0x4160(%ebp)* to point to any memory segments specified by the attacker. In this way, the attacker can rewrite the pointer to a malicious URL after the security check.

The exploit appears to be *GET \x29\xd7\xff\xbf\AAA. . .AA\x40\xf7\xff\xbf\x20\x20/cgi-bin/../../../../bin/sh*, where *0xbfffd729* is just the address of the second part of the URL, and *0xbffff740* is used to overwrite the saved *ebp*. After overwriting the saved *ebp*, the variable addressed by *0x4160(%ebp)*, will be controlled by user input and finally point to */cgi-bin/../../../../bin/sh*.

In this example, deExploit detected eight misuses of input data. The first detected misuse corresponds to the input portion with offset interval [286, 349]. As shown in Fig. 4, this misuse corresponds to a dynamic field that is misused as the value of *esi*. Because this dynamic field is never present in benign executions, we believe that the input portion [286, 349] may be part of its belonging field. In the normalized tree structure, deExploit detected its parent field, which is a string buffer containing the URL and the offset interval is [4, 349]. To localize the root causes of this vulnerability, we find that the address of the memory segment that stores the input portion [286, 349] is *0xbfffb56e*, and the address of the memory segment that stores its belonging field with offset [4, 349] is *0xbfffb426*. Backtracking the memory movement, we can determine that the latest memory operation on the parent field is the memroy movement from *0xbfffb5ad* to *0xbfffb426*, corresponding with the instruction < *rep movsl %ds*: (*%esi*),*%es*: (*%edi*) > . As this memory movement leads to corruption on the memroy region at *0xbfffb56e, 0xbfffb56e* should be the base address of the adjacent data structure. That is, the memory address for the vulnerable data structure may range from *0xbfffb426* to *0xbfffb56e*, with the buffer size no larger than 328.

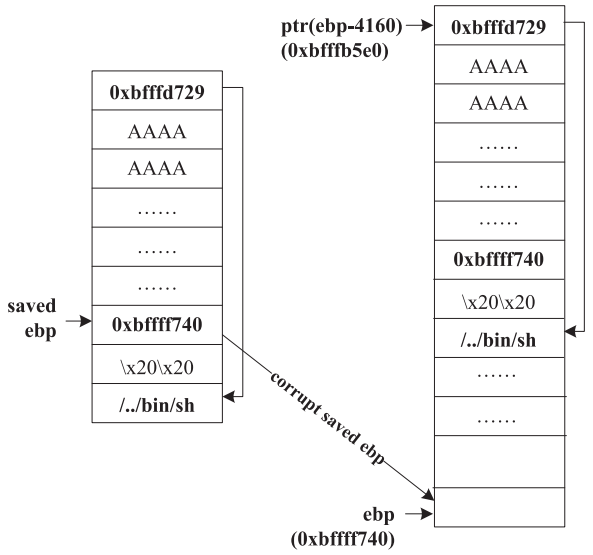


Fig. 6. Attack vector of the data-oriented attack on *ghttpd*.

Actually, the buffer size of the vulnerable data structure *temp* is 200 bytes. The memory segment from *0xbfffb426* to *0xbfffb56e* should contain other data structures than *temp*. To verify the result, we examined the assembly code compiled using debugging symbols and found another buffer named *date_time_final*, which is defined as a buffer with 128 bytes and base address *0xbfffb4ee*. However, the overwritten variable *date_time_final* was never referenced during the execution of the exploit; thus deExploit could not detect it. As deExploit failed to recognize *date_time_final*, the buffer size of the vulnerable memory region was identified by deExploit as being the sum of the buffer *date_time_final* and *temp*.

There are several attack steps during this exploitation, including corrupting the saved *ebp* for stack pivoting, corrupting *ptr* to point to */cgi-bin/../../../../bin/sh*, and executing the command specified by the malicious URL. To highlight these attack steps, deExploit constructed the control-dependency and data-dependency relationships among these misused dynamic fields. As shown in Fig. 7, the exploit first corrupts the memory segment of the saved *ebp* to *0xbfff740*, and the address of this memory cell is *0xbfffb598*. Secondly, *ebp* is used to address another memory cell, i.e., *−0x4160(%ebp)*. The address and value are *0xbfffb5e0* and *0xbfffd729*, respectively. Third, within the execution context of *strncat*, deExploit captures that the pointer *−0x4160(%ebp)* with address *0xbfffb5e0* points to a tainted buffer, which is then copied

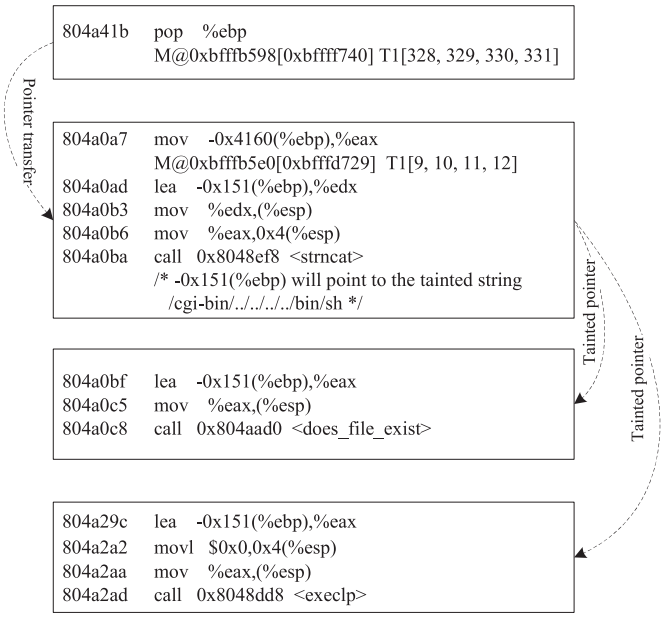


Fig. 7. Key attack steps in the data-oriented attack on *ghttpd*.

to another buffer for which the pointer is *−0x151(%ebp)*. Similarly, by tracking the data flow of *−0x4160(%ebp)* and *−0x151(%ebp)*, deExploit further identifies that *-0x151(%ebp)* is transferred as the parameter to the functions *strcat* and *does_file_exist*, and finally makes the program execute *execlp* with a controlled parameter that points to */cgi-bin/../../../../bin/sh*.

Note that these two data-oriented attacks are not easily detected or diagnosed using previous techniques. For example, in the *gzip* program, the overflow occurs only between two global variables, and leads to no control flow violations. Pointer tainting (Nakka et al., 2005) can detect some data-oriented attacks by detecting tainted pointers, but it cannot detect the exploit of *gzip* because the pointer is taintless. PointerScope (Zhang et al., 2012) diagnoses exploits by identifying conflicts of pointer types. However, in this example, both of the base pointers to *0x807f680* and *0x807faa0* are correctly used, and there is no conflict.

5.2.3. *ROP attack on* coolplayer

As listed in Table 3, for the ROP attack on vulnerability *CVE-2008-3408*, deExploit identified 16 misuses of input data. In this section, we illustrate how these violations can help us understand the attack vector of the exploit.

Fig. 8 shows the trace segment of *coolplayer*, for which deExploit dynamically identified ten dynamic fields. For example, for the instruction *ret*, the instruction address is *0x4089b6* and the operand is accessed through a root pointer addressed by *esp*. Thus, deExploit identifies this operand as a dynamic field. With the taint tags, the dynamic field contains four bytes with offsets ranging from 260 to 263. For the instruction ⟨*pop %ebx*⟩, the instruction address is *0x77f167a6*, and the operand is accessed through another root pointer addressed by *esp*. Thus, deExploit identifies that the dynamic field consists of four bytes with offsets ranging from 264 to 267. Similarly, deExploit identified other eight dynamic fields.

Fig. 8 shows the execution context of these ten dynamic fields. The virtual lines denote the control transfer, and the solid lines denote the data propagation. By examining the context, we find that *eip*, which points to *SetProcessDEPPolicy*, is propagated from the input with offsets ranging from 276 to 279. The argument of *SetProcessDEPPolicy*, which is specified with < *mov* 0x8(*%ebp*),*%eax* >, is propagated from the input with offsets ranging from 264 to 267. By combining the references to these dynamic fields, we can identify the invocation of the function *SetProcessDEPPolicy*, which is used to disable the DEP protection.

5.3. *False positive and false negative*

5.3.1. *False positive*

False positives occur when identified misuses of input data are not attack steps. These misuses are really caused by memory corruption operations, but not for the attack. For example, in the *coolplayer*, we found six misused dynamic fields that do not belong to attack steps. Fig. 9 shows two of the six misused dynamic fields. It is observed that the program pushes two tainted parameters into the stack to call the function at *0x40bf70*. These two parameters

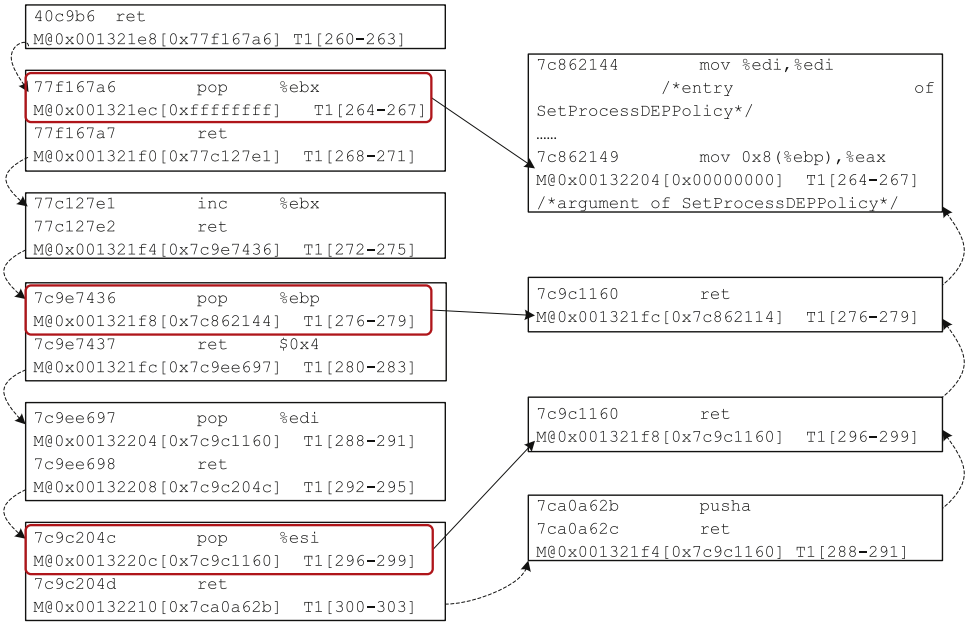


Fig. 8. The execution context of ROP attack on *coolplayer*.

```
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. 9. Misused dynamic fields 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. 10. Source code of the vulnerability *CVE-2006-3747*.

(variables) are actually overwritten by the impact of stack smashing, but they are not attack steps.

To reduce the false positive rate, we propose to capture the dependency relationships between these misused dynamic fields. If a misused dynamic field is not control- or data-dependent on other fields, we claim that this misuse of the dynamic field is not a significant step.

5.3.2. False negative

In our experiments, we encountered one example of a false negative, where deExploit failed to detect the exploit for vulnerability *CVE-2006-3747*. To illustrate this case, we present the source code for the vulnerability in Fig. 10.

This vulnerability is an off-by-one memory corruption. In the code segment, *cp* is defined as a pointer that points to the URL, and the token is defined as an array with five elements. The buggy operation *++c* in line 2740 could lead to an out-of-bounds memory error. Within special scenarios (such as the binary of *Apache-1.3.31* compiled on the Windows platform), the memory region of the token is followed only by the saved return address. This exploit is missed by deExploit. From Fig. 10, it is observed that the data overwriting the saved return address is *cp* itself, instead of the tainted bytes dereferenced by *cp*. Because *cp* is not tainted, deExploit cannot capture this corruption by *cp*. This negative is caused by the under-tainting problem (Schwartz et al., 2010). To solve this problem, we can improve the tainting propagation by enabling control dependencies (Kang et al., 2011).

For the vulnerability caused by integer-overflow-to-buffer-overflow (Zhang et al., 2010), the root causes are integer overflow and buffer overflow. In this type of vulnerability, the length of a buffer is specified by an integer that can be maliciously controlled by the attacker, and then an illegal buffer can further lead to a buffer overflow. To diagnose the exploit for this vulnerability, deExploit can identify the vulnerable data structure, corruption point, and malicious memory operation. However, deExploit may miss some information regarding conditions that must be satisfied to trigger the vulnerability.

6. Discussion

Memory corruption can be caused by over-reading (Szekeres et al., 2013). For example, the well-known heart-bleed vulnerability *heart-bleed (CVE-2014-0160)* in OpenSSL is caused by a memory disclosure attack and does not overwrite any data. Recent advanced attacks, such as JIT-ROP (Snow et al., 2013), also involve memory-disclosure attacks. For exploits where the input data is not misused, we cannot detect the corruption.

Our approach mainly focuses on the memory-corruption vulnerabilities caused by spatial memory errors. For vulnerabilities caused by temporal memory errors (Caballero et al., 2007), it is still possible to identify the misuse of input data. For instance, consider the use-after-free vulnerability in which an invalid pointer dereference can make the program access a deallocated or reallocated memory region. A common exploitation attack on a use-after-free vulnerability is to allocate the new object and range the heap after the old memory object has been deallocated (Lee et al., 2015). The pointer dereference will then misuse the data in the new object as the content of the old object. In the exploit diagnosis where the exploit is available, a much simpler approach to detect the temporal error is to monitor the allocation of the object during the execution of the exploit, as in Memcheck (Seward and Nethercote, 2005).

The widespread use of user scripting and just-in-time compilation allows attackers to carry out an attack practically despite attempts to protect against them (Song et al., 2015). In these advanced exploits, certain memory objects are not directly corrupted by the input data; rather, they are corrupted by the data generated from the just-in-time compilation. In such cases, deExploit may lose its ability to detect the misuse of input data.

7. Related work

Over recent decades, defending against memory corruption has been an ongoing focus of research (Szekeres et al., 2013). In this section, we discuss the most closely related work.

7.1. Reverse engineering

Researchers have proposed many reverse engineering techniques for data structures (DataRescue, 2005), such as ASI (Ramalingam et al., 1999), TIE (Lee et al., 2011), REWARDS (Lin et al., 2010), and Howard (Slowinska et al., 2011). Using static analysis, ASI (Ramalingam et al., 1999) attempts to pinpoint memory segments depending on their access patterns, and can further identify types using the type information obtained from system calls and well-known library functions such as *libc*. Similar to the idea behind ASI, REWARDS (Lin et al., 2010) propagates type information from known parameter types. Howard (Slowinska et al., 2011) constructs symbol tables by capturing the access patterns of different data structures. BYTEWEIGHT (Bao et al., 2014) automatically learns key features to recognize functions.

The reverse engineering in deExploit learns much from previous techniques in terms of capturing the access patterns. Unlike previous techniques such as Howard (Slowinska et al., 2011), which reverses all covered data structures, deExploit only reverses data structures that are used to access input fields. Moreover, tracking

how these data structures are referenced to access the input enables our novel insight of identifying the misuse of input data for the detection of memory-corruption attacks, the localization of the root causes of vulnerabilities, and the identification of the key attack steps in the exploitation attack vector.

7.2. Memory corruption defense

Enforcing memory safety is a generic approach that can stop all types of memory corruption. Representative techniques include pointer bounds and object bounds (Akritidis et al., 2009). Approximation techniques (Akritidis et al., 2008; Castro et al., 2006) that are weaker than memory safety have also been proposed. For example, DFI detects the corruption of any data before they are used by checking read instructions. The safety property is that whenever a value is read, the definition identifier of the instruction that wrote the value is in the reaching definition. The detection policy in deExploit can be substituted by the safety policy in DFI. Compared with DFI, deExploit is more practical for identifying memory corruption at the binary level, whereas DFI requires static analysis of the source code to construct the data-flow graph. For security applications, deExploit is more applicable because it enables security experts who cannot access the source code to perform automatic analysis of exploits and vulnerabilities.

Based on recent studies into reverse engineering, several techniques implement binary-level memory safety or weaker policies by reversing the program data structures. For example, BinArmor (Slowinska et al., 2012) protects data structures with WIT (Akritidis et al., 2008) by reversing the program data structures. StackArmor (Chen et al., 2015) shields binaries from stack-based attacks by recognizing program abstractions such as functions and their control-flow graphs. Data-delineation (Gopan et al., 2015) provides a heuristic analysis for recovering the intended layout of data in stripped binaries, and can be applied to defend against buffer overrun vulnerabilities. These techniques can detect attacks whenever the vulnerabilities are triggered, but they fail to provide precise information about the exploited vulnerability or the attack vector employed in the exploitation.

In recent years, advanced exploits for memory-corruption vulnerabilities have become increasingly sophisticated. JIT-ROP (Snow et al., 2013) was proposed to bypass the protection of both DEP and fine-grained ASLR. Blind-ROP (Bittau et al., 2014) can bypass system protection by adopting a brute-force attack technique. In addition to control-flow-hijack attacks, Hu et al. (2015); 2016) showed that data-oriented exploits could be generated automatically. To defend against memory disclosure attacks, Isomeron (Davi et al., 2015) combined execution-path randomization with code randomization to reduce the success probability of the adversary in predicting the correct runtime address of a target ROP gadget. Execution-only protection techniques such as Execute-no-Read (XnR) (Backes et al., 2014) and Readactor (Crane et al., 2015) have also been proposed to make code pages inaccessible.

To defend against exploits, CFI techniques have been extensively researched in recent years (Zhang and Sekar, 2013; Göktas et al., 2014; Davi et al., 2014), and many practical methods have been proposed (Mohan et al., 2015; van der Veen et al., 2015; Zhang et al., 2013; Payer et al., 2015; Tice et al., 2014). However, these techniques are still designed to detect CFI violations, but they may not work well in defending against data-oriented attacks.

7.3. Crash analysis and exploit diagnosis

For automatic software debugging, researchers have proposed various crash analysis techniques to facilitate the localization of program bugs (Wu et al., 2014; Cui et al., 2016; Jun et al., 2016). Wu et al. (2014) designed CrashLocator to localize software bugs by analyzing stack information in a core dump. Cui et al. (2016) proposed RETracer, a system that reconstructs program semantics from core dumps and examines how the program input contributes to program crashes. RETracer leverages a core dump along with a backward analysis to recover program execution states and spot a software defect. However, CrashLocator and RETracer may lose their capabilities, because their effectiveness relies on the precise analysis of a core dump, whereas the exploitation of vulnerabilities always corrupts memory. In a recent study, CREDAL (Jun et al., 2016) analyzed the core dump of a crashed program without the assumption of memory integrity, and dealt with both corrupted and uncorrupted core dumps to enable crash diagnosis. Compared with these techniques for crash analysis, our approach is more applicable to the security landscape, and enables security experts who cannot access the source code to perform diagnosis and obtain precise information regarding the exploit and vulnerability. More specifically, deExploit can identify the attack vector employed by the exploit, which is valuable for generating exploit signatures; few previous techniques have focused on this problem.

For the automatic localization of vulnerability root causes, researchers have proposed many debugging techniques for memory errors. Memcheck (Seward and Nethercote, 2005) can detect temporal memory errors and memory overlaps; Argos (Portokalidis et al., 2006) tracks the propagation of unsafe data to identify invalid use and generate accurate exploit signatures; and Penumbra (Clause and Orso, 2009) identifies bytes that contribute to an overflow, which is very useful for exploit diagnosis. However, the detection policies of Argos and Penumbra are still based on CFI.

Memsherlock (Sezer et al., 2007) automatically identifies unknown memory-corruption vulnerabilities and provides critical information on corruption points in the source code to describe how the malicious input exploits the unknown vulnerability. The problem scope of deExploit is identical to that of Memsherlock. The difference is that deExploit works at the binary level, whereas Memsherlock requires the source code to be rewritten. The binary-level approach is more practical, because many security analyses are performed by third-parties who cannot access the source code or debugging information.

Dependency-based techniques can trace back to the source data after detecting attacks (Newsome and Song, 2005), but cannot clearly distinguish key attack steps and other dependencies.

Execution comparison is widely used for software debugging and security analysis (Sumner et al., 2011), which are closely related to our study. Differential slicing (Johnson et al., 2011) isolates the causal path of state differences that lead to the observed difference. Dual slicing (Weeratunge et al., 2010) compares two executions and extracts the differing dependencies between them to debug concurrency bugs. The main challenge to existing execution comparisons is that there may be a large number of different program states. Our approach does not select the detailed program states for comparison. Instead, deExploit models the way the program manipulates benign input as a criterion for detecting the invalid references to the exploit.

8. Conclusion

Exploit diagnosis is an important yet time-consuming task in software security defense techniques. This paper proposed the deExploit tool, a novel binary-level approach for exploit diagnosis, for detecting memory corruption by identifying the misuse of inputs. Evaluations on several realistic exploits show that deExploit is a generic technique for attack detection, including both control-hijacking and data-oriented exploit, and is effective in localizing the root causes and identifying key attack steps.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (nos. 61303213, 61373169, 61672394), and National High Technology Research and Development Program of China (No. 2015AA016004).

References

Abadi, M., Budiu, M., Erlingsson, Í, Ligatti, J., 2009. Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. 13 (1), 1–40.

Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M., 2008. Preventing memory error exploits with WIT. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy, pp. 263–277.

Akritidis, P., Costa, M., Castro, M., Hand, S., 2009. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the 18th conference on USENIX Security Symposium, pp. 51–66.

Backes, M., Holz, T., Kollenda, B., Koppe, P., Nürnberger, S., Pewny, J., 2014. You can run but you can't read: preventing disclosure exploits in executable code. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14, pp. 1342–1353.

Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D., 2014. Byteweight: learning to recognize functions in binary code. In: Proceedings of the the 23rd USENIX Security Symposium.

Bhatkar, S., DuVarney, D., Sekar, R., 2003. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12th USENIX Security Symposium, pp. 105–120.

Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., Boneh, D., 2014. Hacking blind. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy, pp. 227–242.

Brumley, D., Newsome, J., Song, D., 2006. Towards automatic generation of vulnerability-based signatures. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy. IEEE, pp. 15–30.

Buchanan, E., Roemer, R., Shacham, H., Savage, S., 2008. When good instructions go bad: generalizing return-oriented programming to risc. In: Proceedings of the 15th ACM conference on Computer and communications security - CCS '08, pp. 27–38.

Caballero, J., Yin, H., Liang, Z., Song, D., 2007. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the 14th ACM conference on Computer and communications security, pp. 317–329.

Carbin, M., Rinard, M.C.M., 2010. Automatically identifying critical input regions and code in applications. In: Proceedings of the 19th international symposium on Software testing and analysis. ACM, New York, New York, USA, pp. 37–48.

Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R., 2015. Control-flow bending: on the effectiveness of control-flow integrity. In: Proceedings of the 24th USENIX Conference on Security Symposium, pp. 161–176.

Castro, M., Costa, M., Harris, T., 2006. Securing software by enforcing data-flow integrity. In: Proceedings of the 7th symposium on Operating systems design and implementation, pp. 147–160.

Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K., 2005. Non-control-data attacks are realistic threats. In: Proceedings of the 14th Conference on USENIX Security Symposium.

Chen, X., Slowinska, A., Andriesse, D., Bos, H., Giuffrida, C., 2015. StackArmor: comprehensive protection from stack-based memory error vulnerabilities for binaries. In: Proceedings of the 2015 Network and Distributed System Security Symposium.

Clause, J., Doudalis, I., Orso, A., Prvulovic, M., 2007. Effective memory protection using dynamic tainting. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07. ACM Press, New York, New York, USA, pp. 284–292.

Clause, J., Orso, A., 2009. Penumbra: automatically identifying failure-relevant inputs using dynamic tainting. In: Proceedings of the eighteenth international symposium on Software testing and analysis, pp. 249–260.

Costa, M., Castro, M., Zhou, L., Zhang, L., Peinado, M., 2007. Bouncer: securing software by blocking bad input. In: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pp. 117–130.

Cowan, C., Beattie, S., Johansen, J., 2003. Pointguard: protecting pointers from buffer overflow vulnerabilities. In: Proceedings of the 12th USENIX Security Symposium, pp. 91–104.

Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., 1998. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th conference on USENIX Security Symposium.

Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A.-R., Brunthaler, S., Franz, M., 2015. Readactor: practical code randomization resilient to memory disclosure. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy.

Cui, W., Peinado, M., Cha, S.K., Fratantonio, Y., Kemerlis, V.P., 2016. RETracer: triaging crashes by reverse execution from partial memory dumps. In: Proceedings of the 38th International Conference on Software Engineering - ICSE '16. ACM Press, New York, New York, USA, pp. 820–831. doi:10.1145/2884781.2884844.

Cui, W., Peinado, M., Chen, K., Wang, H., Irun-Briz, L., 2008. Tupni: automatic reverse engineering of input formats. In: Proceedings of the 15th ACM conference on Computer and communications security - CCS '08, pp. 391–402.

Data Execution Protection, 2013. https://support.microsoft.com/en-us/kb/875352.

DataRescue, 2005. High level constructs width IDA Pro.

Davi, L., Liebchen, C., Sadeghi, A.-R., Snow, K.Z., Monrose, F., 2015. Isomeron: code randomization resilient to (just-in-time) return-oriented programming. In: Proceedings of the 2015 Network and Distributed System Security Symposium.

Davi, L., Sadeghi, A.-R., Lehmann, D., Monrose, F., 2014. Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In: Proceedings of the 23rd USENIX Security Symposium, pp. 401–416.

Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., Sidiroglou-Douskos, S., 2015. Control jujutsu: on the weaknesses of fine-grained control flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15. ACM Press, New York, New York, USA, pp. 901–913. doi:10.1145/2810103.2813646.

Gkantsidis, C., Karagiannis, T., VojnoviC, M., 2006. Planet scale software updates. In: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New York, USA, pp. 423–434.

Godefroid, P., Kiezun, A., Levin, M.Y., 2008. Grammar-based whitebox fuzzing. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '08, 43, pp. 206–215.

Göktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G., 2014. Out of control: overcoming control-flow integrity. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy.

Gopan, D., Driscoll, E., Nguyen, D., Naydich, D., Loginov, A., Melski, D., 2015. Data-delineation in software binaries and its application to buffer-overrun discovery. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1.

Haller, I., Slowinska, A., Bos, H., Neugschwandtner, M.M., 2013. Dowsing for overflows: a guided fuzzer to find buffer boundary violation. In: Proceedings of the 22nd USENIX Security Symposium, pp. 49–64.

Hu, H., Chua, Z.L., Adrian, S., Saxena, P., Liang, Z., 2015. Automatic generation of data-oriented exploits. In: Proceedings of the 24th USENIX Security Symposium.

Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z., 2016. Data-oriented programming: on the expressiveness of non-control data attacks. In: Proceedings of the 2016 IEEE Symposium on Security and Privacy. IEEE, pp. 969–986.

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

Jun, X., Dongliang, M., Ping, C., Xinyu, X., Pei, W., Peng, L., 2016. CREDAL: towards locating a memory corruption vulnerability with your core dump. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS '16.

Kang, M.G., McCamant, S., Poosankam, P., Song, D., 2011. Dta++: dynamic taint analysis with targeted control-flow propagation.. In: Proceedings of the 18th Annual Network and Distributed System Security Symposium.

Kil, C., Sezer, E., Ning, P., Zhang, X., 2007. Automated security debugging using program structural constraints. In: Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007). IEEE, pp. 453–462.

Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., Lee, W., 2015. Preventing use-after-free with dangling pointers nullification. In: Proceedings of the 22nd Annual Network and Distributed System Security Symposium.

Lee, J., Avgerinos, T., Brumley, D., 2011. TIE: principled reverse engineering of types in binary programs. In: Proceedings of the 2011 Network and Distributed System Security Symposium.

Lin, Z., Jiang, X., Xu, D., Zhang, X., 2008. Automatic protocol format reverse engineering through context-aware monitored execution. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium.

Lin, Z., Zhang, X., Xu, D., 2010. Automatic reverse engineering of data structures from binary execution. In: Proceedings of the 17th Network and Distributed System Security Symposium.

Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K., Franz, M., 2015. Opaque control-flow integrity. In: Proceedings of the 2015 Network and Distributed System Security Symposium.

Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., Weaver, N., 2003. Inside the slammer worm. IEEE Secur. Privacy Mag. 1 (4), 33–39.

Nakka, N., Kalbarczyk, Z., Iyer, R., 2005. Defeating memory corruption attacks via pointer taintedness detection. In: 2005 International Conference on Dependable Systems and Networks (DSN'05), pp. 378–387.

Newsome, J., Song, D., 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the 12nd Annual Network and Distributed System Security Symposium.

Offensive security exploit database archive, 2016. https://www.exploit-db.com/.

PanguTeam, 2015. The Userland Exploits of Pangu 8. Technical Report.

Payer, M., Barresi, A., Gross, T.R., 2015. Fine-grained control-flow integrity through binary hardening. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 144–164.

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

Prakash, A., Yin, H., Liang, Z., 2013. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. ACM Press, New York, USA, pp. 311–322.

Ramalingam, G., Field, J., Tip, F., 1999. Aggregate structure identification and its application to program analysis. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 119–132.

Schwartz, E.J.E., Avgerinos, T., Brumley, D., 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. IEEE, pp. 317–331.

Seward, J., Nethercote, N., 2005. Using valgrind to detect undefined value errors with bit-precision. In: Proceedings of the 2005 USENIX Conference on Annual Technical Conference, pp. 17–30.

Sezer, E.C., Ning, P., Kil, C., Xu, J., 2007. Memsherlock: an automated debugger for unknown memory corruption vulnerabilities. In: Proceedings of the 14th ACM Conference on Computer and Communications Security - CCS '07. ACM Press, New York, USA, p. 562.

Slowinska, A., Stancescu, T., Bos, H., 2011. Howard: a dynamic excavator for reverse engineering data structures. In: Proceedings of the 2011 Network and Distributed System Security Symposium.

Slowinska, A., Stancescu, T., Bos, H., 2012. Body armor for binaries: preventing buffer overflows without recompilation. In: Proceedings of the 2012 USENIX conference on Annual Technical Conference. 11–11.

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

Song, C., Zhang, C., Wang, T., Lee, W., Melski, D., 2015. Exploiting and protecting dynamic code generation. In: Proceedings of the 2015 Network and Distributed System Security Symposium.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P., 2008. BitBlaze: a new approach to computer security via binary analysis. In: the 4th International Conference on Information Systems Security.

Subverting without EIP, 2014. http://mallocat.com/subverting-without-eip/.

Suh, G.E., Lee, J.W., Zhang, D., Devadas, S., 2004. Secure program execution via dynamic information flow tracking. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 85–96.

Sumner, W., Bao, T., Zhang, X., 2011. Selecting peers for execution comparison. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 309–319.

Szekeres, L., Payer, M., Wei, T., Song, D., 2013. SoK: eternal war in memory. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy, pp. 48–62.

Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G., 2014. Enforcing forward-edge control-flow integrity in gcc & llvm. In: Proceedings of the 23rd USENIX Security Symposium, pp. 941–955.

van der Veen, V., Andriesse, D., Gökta, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., Giuffrida, C., 2015. Practical context-sensitive CFI. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15. ACM Press, New York, USA, pp. 927–940.

Veen, V.V.D., Dutt-Sharma, N., Cavallaro, L., Bos, H., 2012. Memory errors: the past, the present, and the future. In: Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses, pp. 86–106.

Weeratunge, D., Zhang, X., Sumner, W.N., Jagannathan, S., 2010. Analyzing concurrency bugs using dual slicing. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, pp. 253–264.

Wu, R., Zhang, H., Cheung, S.-C., Kim, S., 2014. CrashLocator: locating crashing faults based on crash stacks. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014. ACM Press, New York, USA, pp. 204–214.

Xu, J., Ning, P., Kil, C., Zhai, Y., Bookholt, C., 2005. Automatic diagnosis and response to memory corruption vulnerabilities. In: Proceedings of the 12th ACM Conference on Computer and Communications Security - CCS '05, pp. 223–234.

Yong, S.H., Horwitz, S., 2003. Protecting C programs from attacks via invalid pointer dereferences. In: Proceedings of the 9th European Software Engineering Conference Held Jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering - ESEC/FSE '03. ACM Press, New York, New York, USA, pp. 307–316.

Zhang, C., Wang, T., Wei, T., Chen, Y., Zou, W., 2010. IntPatch: automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In: Proceedings of he 2010 European Symposium on Research in Computer Security, pp. 71–86.

Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W., 2013. Practical control flow integrity & randomization for binary executable. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy, pp. 559–573.

Zhang, M., Prakash, A., Li, X., Liang, Z., Yin, H., 2012. Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis. In: Proceedings of the 19th Annual Network and Distributed System Security Symposium.

Zhang, M., Sekar, R., 2013. Control flow integrity for COTS binaries. In: Proceedings of the 22nd USENIX conference on Security, pp. 337–352.

Zhao, L., Wang, R., Wang, L., Cheng, Y., 2015. Reversing and identifying overwritten data structures for memory-corruption exploit diagnosis. In: 2015 IEEE 39th Annual Computer Software and Applications Conference, pp. 434–443.

Run Wang, a Ph.D candidate in the State Key Laboratory of Software Engineering in Wuhan University, China. His research focuses on software testing and debugging.

Pei Liu, a software engineer in Changjiang River Scientific Research Institute, China. Her research mainly focuses on software automation and software analysis.

Lei Zhao, an associate professor in the State Key Laboratory of Software Engineering in Wuhan University, China. He got his Doctors Degree in 2012, worked as an assistant professor in Wuhan University from 2013 to 2015. His research mainly focuses on software security, software analysis, and system protection.

Yueqiang Cheng, a software engineer in APL Software, USA. His research mainly focuses on system dependability and security.

Lina Wang, a professor in the State Key Laboratory of Software Engineering in Wuhan University, China. Her research mainly focuses on system security, multimedia security and cloud computing.