On-Demand Proactive Defense against Memory Vulnerabilities

Gang Chen, Hai Jin, Deqing Zou, and Weiqi Dai

Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
hjin@hust.edu.cn

Abstract. Memory vulnerabilities have severely affect system security and availability. Although there are a number of solutions proposed to defense against memory vulnerabilities, most of existing solutions protect the entire life cycle of the application or survive attacks after detecting attacks. This paper presents OPSafe, a system that make applications safely survive memory vulnerabilities for a period of time from the starting or in runtime with users' demand. OPSafe can provide a hot-portable *Green Zone* of any size with users' demand, where all the subsequent allocated memory objects including stack objects and heap objects are reallocated and safely managed in a protected memory area. When users open the green zone, OPSafe uses a comprehensive memory management in the protected memory area to adaptively allocate buffers with multiple times of their defined sizes and randomly place them. Combined with objects free masking techniques, OPSafe can avoid overrunning each other and dangling pointer errors as well as double free or invalid free errors. Once closing the green zone, OPSafe clears away all objects in the protected area and then frees the protected area. We have developed a Linux prototype and evaluated it using four applications which contains a wide range of vulnerabilities. The experimental results show that OPSafe can conveniently create and destruct a hot-portable green zone where the vulnerable application can survive crashes and eliminate erroneous execution.

Keywords: Memory Vulnerabilities; Proactive Defense.

1 Introduction

Memory bugs severely affect system security and availability. Programs written in unsafe languages like C and C++ are particularly vulnerable because attackers can exploit memory errors to control vulnerable programs. These vulnerabilities can also cause programs failures. According to a survey conducted by IT industry analyst firms [10], the average business loss of an hour of IT system downtime is between US $84,000 and US $108,000. However, previous study showed that the average time to diagnose bugs and generate patches is 28 days. During this long

C.-H. Hsu et al. (Eds.): NPC 2013, LNCS 8147, pp. 368–379, 2013.

time vulnerable window, users have to either stop running the program which cause the costly program downtime, or continuing running the program which experience problems such as potential crashes and attacks. Neither of the two behaviors is desirable. Therefore, it is significant to survive attacks to preserve system availability.

There are a number of solutions have been proposed to protect programs against memory vulnerabilities. One direction is for buffer overflow prevention, including return address defense on the stack [4,6], array bounds checking, pointers protection via encrypting pointers, and address obfuscation via randomizing memory layout [2]. These approaches can improve the safety while drop the availability. The reason is that programs are always terminated when detecting attacks. The other direction is for heap-related bug prevention [1,5], which well balances the safety and availability of the applications. However, these techniques cannot be applied for protecting against stack buffer overflow attacks.

Previous solutions on protecting against memory vulnerabilities had to protect the entire life cycle of the application, or survive attacks after detecting attacks. However, users only need protection for a period of time at most times, especially for desktop applications. For example, when users use a browser to submit works, they need the browser not occurring any faults until the works have been submitted successfully.

In this paper, we propose a on-demand proactive protection system called OPSafe, which can make applications safely survive memory vulnerabilities for a period of time from the starting or in runtime with users' demand. When users need to protect the program against memory vulnerabilities, OPSafe can dynamically reallocate all the subsequent allocated memory objects in an protected memory area, called *Green Zone*, and it can also conveniently break away from the program to terminate the protection when users will not protect the program. In the green zone, OPSafe adopts the memory randomization technique in the protected memory area, which adaptively allocates buffers with multiple times of their defined sizes and places semantically infinite distance between memory objects to provide an probabilistic memory safety. This technique can greatly avoid overrunning between memory objects as well as dangling pointer errors. By omitting invalid free, OPSafe can mask double-free and invalid free errors. When closing green zone, OPSafe clears away the objects in the protected area and then frees the protected area. Our green zone is hot-portable with a feature of conveniently creating and destructing for running applications. As a result, OPSafe can provide a hot-portable green zone of any size for an application with users' demand.

To demonstrate the effectiveness of OPSafe, we have implemented a Linux prototype and evaluated it using four applications that contain a wide range of memory vulnerabilities including stack smashing, heap buffer overflow, double-free and dangling pointer. Our experimental results show that OPSafe can protect applications against memory vulnerabilities in the green zone with a reasonable overhead.

In summary, we make the following contributions.

- We propose OPSafe, an on-demand proactive protection system to make applications safely survive memory bugs for a period of time from the starting or in runtime with users' demand.
- We propose a concept, a hot-portable green zone of any size with users' demand, which can conveniently protect the applications at any times. It can protect the applications from the beginning or in runtime, and slice away from applications at any times, both decided by users.
- We have implemented OPSafe on a Linux system, and evaluated its effectiveness and performance using four applications that contain a wide ranges of memory vulnerabilities.

The rest of the paper is organized as follows. The OPSafe overview, including the motivation, the background, the architecture, the workflow and important steps of OPSafe, are introduced in section 2. Section 3 describes the important implementation techniques. The experimental and analytical results are presented in section 4. Section 5 gives an overview of related work. Finally, we summarize our contributions in section 6.

2 OPSafe Overview

In this section, we first introduce our motivation, and then give an overview of the architecture and workflow of OPSafe.

2.1 Motivation

Memory vulnerabilities have severely affected system security and availability. Full life-cycle protection for an application is common most of times. However, sometimes we may need the application to be secure and reliable only for a period of time, especially for desktop applications. For example, when we write an important email via an email client, we need the client not crashed in the writing. However, we may not mind crashing in normal use because restarting the client is also convenient. Moreover, we even do not want to use the protection tool to protect the client in normal use because they affect the use of the client more or less from the performance or the function.

A great number of solutions on handling with memory vulnerabilities have been proposed, but they are designed from the view on how to respond to attacks without considering the users' demand. They can be classified into three categories: proactive full life-cycle protections which protect the application from the starting to the termination, fail-stop approaches which terminate the application for security when detecting a fault, and self-healing approaches which learn from the fault and automatically temporarily fix the bugs. All of these solutions should always monitor the application and not conveniently slice away from the application.

Take the email client as an example again. A user wants a green zone and also can conveniently control the size of the green zone for the email client. The green zone can only be opened with the user demand. When writing the important email, the user can open the green zone. In this green zone, the user does not mind using more resource to make sure the security and availability of the application. When the resource occupied by the protection tool is needed to do other things, the user can conveniently close the green zone. Therefore, we need an on-demand proactive defense mechanism which can safely and effectively survive memory vulnerabilities in the green zone and have no effect outside of it for an application.

The memory management of OPSafe is based on our previous work SafeStack [11] and Memshepherd [12]. SafeStack proposes the memory access virtualization mechanism to reallocate stack objects into a protected memory area. Memshepherd integrates the memory access virtualization mechanism into DieHard's design on the probabilistic safe heap memory allocator to reallocate heap objects into a safe heap space. In this heap space, memory objects are randomly placed in large chunks called *miniheaps*. However, Memshepherd should manage all the memory objects from the starting of the application, which can incur a high overhead and make the performance of the application degrade.

2.2 OPSafe Architecture

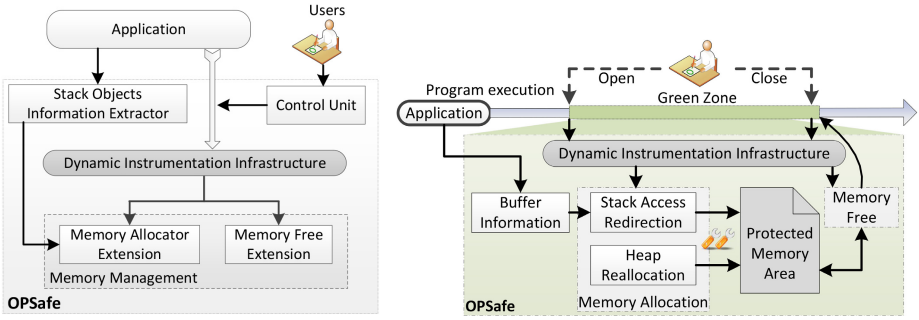
The architecture of OPSafe is illustrated in Figure 1a. OPSafe consists of several components, including a Stack Objects Information Extractor used to extract the information of stack buffers from the binaries, a Control Unit used to respond users' demand to generate a green zone (i.e., a protection window to protect and to end protection for the application), a Memory Allocator Extension used to reallocate all memory objects including stack objects and heap objects into the protected memory area, and a Memory Free Extension used to free the protected memory area.

Based on a dynamic instrumentation infrastructure, OPSafe can generate a hot-portable green zone for the application, including allocating and freeing memory objects. To reallocate the stack objects into the protected memory areas, OPSafe should get the location and size information of the original stack objects, which is gained by the static analysis from the Stack Objects Information Extractor.

2.3 Workflow of OPSafe

The workflow of OPSafe is illustrated in Figure 1b. Users can arbitrarily open or close the green zone on demand. The stack buffer information can be extracted from binaries before opening the green zone.

When users request to open the green zone, OPSafe creates a protected memory area and uses the Memory Allocator Extension to dynamically reallocate all the memory objects into the protected memory area with the dynamic instrumentation infrastructure. The memory allocation contains two parts. One is



(a) Architecture (b) Procedure

Fig. 1. OPSafe Design

stack access redirection which is the basis of memory access virtualization. It is used to reallocate the stack objects into the protected memory area and redirect the memory access from the original address to the corresponding protected memory address. In this redirection, OPSafe should locate the original address of stack objects which can be gained from the buffer information. The other is heap reallocation which is used to intercept the heap allocation functions and replace them with our allocation functions to make all the heap objects be allocated in the protected memory area. In the protected memory area, OPSafe maintains stack objects of a function until the function exits and marks the the memory area of an heap object can be used when the application frees it.

When users request to close the green zone, OPSafe uses the Memory Free Extension to restore the memory management of the application and gradually frees all the memory objects in the protected memory area. OPSafe firstly removes all the reallocation instruments for stack objects and heap objects reallocation. In addition, OPSafe checks all *miniheaps* to free the unused *miniheaps*. If there are *miniheaps* in use, OPSafe then continues to monitor them. Once all the rest of *miniheaps* have been freed, OPSafe destructs the green zone by removing all the free operation instruments and gracefully slicing away from the application. After that, the green zone is closed.

3 OPSafe Implementation

In our implementation, our protected memory area is a heap space which is managed in the same way as Memshepherd. We should note that our protected memory area is not tied with Memshepherd's memory management, but also can be combined with other safe memory management mechanisms. In this section, we discuss the stack buffer information extraction, the green zone creation, maintenance and destruction.

3.1 Stack Buffer Information Extraction

The stack buffer information consists of the function level information and the variable level information. The function level information contains the function name, the starting code address and the number of stack buffer variables. The variable level information contains the variable name, size, offsets from the frame pointers for each local variables and parameters for buffers. As Memshepherd extracts stack buffer information according to the memory objects access pattern, the information is not accurate. However, if the program is compiled with debugging option (*-g* in GCC), the compiler adds debugging information about all variables in the program binaries. We extend Memshepherd's buffer information extractor with debugging information for the applications compiled with debugging option.

OPSafe uses TIED [13] with an extension to extract debugging information for stack buffers. TIED cannot extract the stack buffer information from function *main* as the offset is relative to the stack pointer. OPSafe organizes all the buffer related information according to TIED, but makes some changes to the data structure of the buffer related information for future effective diagnoses, and also OPSafe does not need to rewrite the program binaries with the buffer related information as TIED does.

3.2 Green Zone Creation

When users request to open the green zone, OPSafe uses the dynamic instrumentation tool to intercept the memory allocation operations, including stack objects allocation and heap objects allocation. In addition, OPSafe creates the protected memory area which is allocated adaptively according to usage requirements.

For stack objects allocation, OPSafe only redirects the stack buffers in the subsequent called functions into the protected memory area. To redirect these stack buffers access, OPSafe should firstly locate the original addresses of stack buffers according to the stack buffer information, and then map the original addresses to the new corresponding addresses in the protected area. This is done when the register *ESP* becomes smaller. As the stack buffers have not been initiated at that time, OPSafe can benefit from avoiding copying data from the original memory addresses to the new corresponding addresses. For the stack buffers in the current stack, OPSafe maintains the original memory access to these stack buffers.

For heap objects allocation, OPSafe intercepts all the heap allocation functions in *libc* and replaces them with OPSafe's heap allocation functions, such as the function *malloc*. OPSafe only reallocates the subsequent heap objects requests after green zone creation, and the management for the previous allocated heap objects is still maintained by the application.

3.3 Green Zone Maintenance

After memory objects reallocation, OPSafe maintains the life cycle of these memory objects in the green zone. For the memory objects outside of the green zone, OPSafe leaves them into the management of the application.

For the stack buffers access, there are three types of access for them in the memory access virtualization, including direct access, indirect access and pointer access. The access mode is "*Base plus Offset*" for direct access and "*Base plus Index plus Offset*" for indirect access. The base is the base register, i.e., *EBP* or *ESP*, the offset is the offset is the value between the starting address of the array and the base register, and the index is the index register which stores the stack buffer index. For these two cases, OPSafe calculates the starting address of the stack buffer and replaces with the corresponding protected memory address. The pointer access means stack buffer is accessed by pointers, such as a stack buffer is passed an argument to a function. As there must be an instruction to get the address of the stack buffer, OPSafe can replace it without continuing to map the memory access from the pointer. When a function returns, OPSafe frees all the stack buffers of the function in the protected memory area.

For the heap object access, there are no extra memory mapping operations. When freeing a heap object, OPSafe checks whether it is in the green zone or not. If so, OPSafe then checks whether the heap object has been freed or the address of the heap object is invalid. For the double-free or invalid free, OPSafe can mask this error by omitting these memory free operations. If the heap object is outside of the protected memory area, OPSafe leaves the management of these objects and make the application to call the original *libc* function *free* to free it.

3.4 Green Zone Destruction

When users request to close the green zone, OPSafe enters into the green zone destruction phrase to gradually free all the memory objects in the green zone and restore the memory management of the application.

In this phrase, OPSafe firstly checks all the *miniheaps* in the green zone to determine which the *miniheap* can be freed. If there are *miniheaps* which are unused (i.e., do no store any memory objects), OPSafe frees these *miniheaps*. For the *miniheaps* which are in use, OPSafe should wait until all the memory objects of a *miniheap* have been freed. The memory objects in a miniheap consists of stack objects and heap objects. OPSafe frees stack objects of a function when the function returns and frees heap objects when the application call the *libc* function *free* to free them. In addition, OPSafe does not need to redirect the subsequent stack objects of new called functions and intercept the memory allocation operations of the application.

Finally, when all the *miniheaps* has been freed, OPSafe destructs the green zone by removing all the instrumentation for the application to restore the memory management of the application. After that, the application runs as original.

Table 1. Applications and bugs used in evaluation

Application	Version	Vulnerabilities ID	Bug	Description
ProFTPD	1.3.3b	CVE-2010-4221	Stack Overflow	FTP Server
Null-HTTPd	0.5.0	CVE-2002-1496	Heap Overflow	Web Server
Null-HTTPd-df		Manually Injected	Double Free	
Pine	4.44	CVE-2002-1320	Heap Overflow	Email Client
M4	1.4.4	-	Dangling Pointer	Macro Processor

4 Experimental Evaluation

We have implemented a Linux prototype system with the operating system kernel Linux 2.6, and Pin 2.12-56759 is used as our dynamic instrumentation infrastructure. Our evaluation platform consists of two machines connected with 100Mbps Ethernet. One machine is configured with the Intel E5200 dual core 2.5GHz processors and 4GB memory. They are used to deploy OPSafe and our test suite. The other is configured with Intel E7400 dual core 2.8GHz processors and 4GB memory. It is used to run clients for testing servers and servers for testing desktop clients.

We first describe the details of our test suite which are a range of multi-process and multi-threaded applications. Then we present the results of the evaluation on the effectiveness of OPSafe under our test suite. To test the effectiveness, we use OPSafe to protect our test suite to survive memory vulnerabilities. Finally, we discuss the performance evaluation.

4.1 Overall Analysis

Our test suite contains four applications, including a ftp server ProFTPD, a web server Null-HTTPd, an email client Pine and a macro processor GNU M4, which are listed in Table 1. All these applications contain various types of bugs, including stack overflow, heap overflow, double free and dangling pointer. Four of the vulnerabilities are real-world vulnerabilities, and the double free bug is manually injected into the web server Null-HTTPd, named Null-HTTPd-df in Table 1.

Exploit CVE-2010-4221 in ProFTPD enables attackers corrupt memory to crash the ftp server or execute arbitrary code by sending data containing a large number of Telnet IAC commands to overrun stack buffers in the vulnerable function *pr_cmd_read*. OPSafe can avoid overrunning the control structures in the stack by moving the stack buffers of the vulnerable function into the protected memory area.

Exploit CVE-2002-1496 in Null-HTTPd is caused by improper handling of negative *Content-Length* values in HTTP header field. By sending a HTTP request with a negative value in the *Content-Length* header field, a remote attacker

could overflow a heap buffer and cause the web server to crash or execute arbitrary code in the system. OPSafe can avoid attackers gain control by segregating all the heap metadata from the heap in the protected memory area. In addition, OPSafe can avoid the vulnerable heap objects overrunning other heap objects with high probability by multiple times of their defined sizes and a random placement strategy.

Exploit manually injected in Null-HTTPd-df is caused by a piece of injected vulnerable code. It frees a heap object which has already been freed. OPSafe can mask this double free error by intercepting the free operation, checking it and omitting invalid free operation.

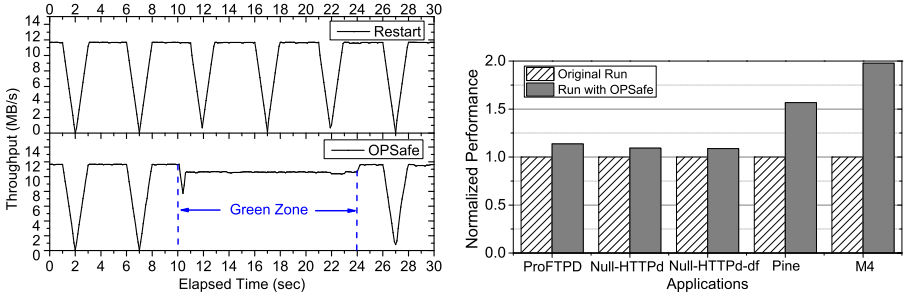
Exploit CVE-2002-1320 in Pine enables attackers send a fully legal email message with a crafted From-header to force Pine to core dump on start-up. The only way to launch pine is manually removing the bad message either directly from the spool, or from another mail user agent. Until the message has been removed or edited there is no way of accessing the INBOX using Pine. The heap overflow is caused by the incorrect calculation of string length in the function *est_size*, a message's header "*From:*" which contains a long string of escaped characters can cause a buffer used by the function *addr_list_string* to overflow. OPSafe can allocate an object with multiple times of defined size and a random placement strategy to avoid overrunning with high probability.

Exploit the dangling pointer in GNU M4 causes an misbehavior such as printing out misleading information when a macro whose arguments are being collected is redefined or deleted. Through deleting the definition from the symbol table, it can leave dangling pointers in the local variable *sym* of the function *expand_macro* and then use dangling pointers leading M4 misbehavior. OPSafe can avoid this misbehavior as it does not really free a heap object in the green zone. It holds the content of the object and only marks the memory space be used. Fortunately, the random placement strategy of OPSafe can make it unlikely that a newly freed object will soon be overwritten by a subsequent allocation. Therefore, OPSafe can avoid this dangling pointer attacks with high probability.

4.2 Effectiveness Evaluation

We evaluate the capability of fault tolerance by comparing OPSafe with the restart method. We use a lightweight web server Null-HTTPd in this experiment. We adopte the apache benchmark *ab* to test the web server throughput and a network traffic and bandwidth usage tool *nload* to get a real time throughput with a $100ms$ interval. In addition, malicious requests are sent every $5s$ to crash the web server. All these two cases have adopted the restart method which restarts the web server when it crashes, and the results are shown in Figure 2a.

From the figure, we can see that OPSafe can survive the heap overflow attacks to make the web server continually serve the clients in the green zone with a modest performance degradation. Once closing the green zone, the web server still crashes when attacks occurring.



(a) Comparison between Restart and OPSafe (b) Overhead for Normal Execution

Fig. 2. OPSafe Evaluation

4.3 Performance Evaluation

We evaluated the normal execution overhead caused by OPSafe with these four applications. In this experiment, we use a ftp client to download a 300MB file from the ProFTPD server. For the web server Null-HTTPd, we use the Apache web benchmark *ab* to get the result. For the Email client Pine, we use it to send an email whose size is 9.5MB. For the GNU M4, we use its example test file *foreach.m4* as the workload. We compared the average response time for server applications and the execution time for desktop applications.

We show the overhead of OPSafe in Figure 2b. It ranges from 8.77% for Null-HTTPd-df to 97.87% for GNU M4 with an average of 37.28%. Although OPSafe incurs a modest overhead, it provides a flexible protection mechanism with users' demand.

5 Related Work

In this section, we compare OPSafe with other solutions in preventing and responding to memory vulnerabilities attacks.

5.1 Proactive Defense Methods

Failure-oblivious computing [8] tolerates memory vulnerabilities via manufacturing values for "out-of-bound read" and discarding "out-of-bound write". However, it should modify the source codes of the application with its specific compiler. Moreover, it may cause an unpredicted behavior which is an new threat for the application.

DieHard [1] adopts the randomized allocation technique to give the application an approximation of an infinite sized heap, which can provide an high probabilistic memory safety. However, it should protect the application from the beginning and it is incapable for surviving stack smashing attacks.

5.2 Fail-Stop Methods

There are a number of solutions proposed to defend against memory vulnerabilities in a fail-stop fashion, especially for stack smashing bugs. StackGuard [4] checks the integrity of canaries which are inserted around the return address. Address space layout randomization technique is proposed to protect applications against memory bugs exploits with a source-to-source transformation [2]. These solutions have to use the compiler to analyze or modify source codes via extending the GNU C compiler. However, OPSafe can protect applications without extending the compiler.

A binary rewriting defense technology [6] is proposed to protect the return address. It inserts the return address defense codes into binaries of applications which can protect the integrity of the return address with a redundant copy. However, it is powerless for heap-related vulnerabilities.

5.3 Self-healing Methods

Rx [7] combines the checkpoint/rollback mechanism and a changing execution environment to survive bugs. Especially for the stack smashing bugs, Rx drops the malicious users' requests.

ASSURE [9] proposes the error virtualization and rescue point technique to bypass faulty region of codes. ASSURE may cause the application not function well, our previous works SHelp [3] applies weighted rescue points and extends it to a virtualization computing environment. However, all of these techniques unsafely speculate on programmer's intentions which may introduce new threats.

First-Aid [5] tolerates a bug via identifying the bug types and bug-triggering memory objects, and generating runtime patches to apply them to a small set of memory objects. However, it cannot deal with the stack smashing bugs.

6 Conclusion

In this paper we present a new system OPSafe which can provide a hot-portable *Green Zone* of any size with users demand for applications where applications can safely survive memory bugs for a period of time from the starting or in runtime with users' demand. Once the green zone is opened, OPSafe takes over memory management of the application to make all the subsequent allocated memory objects including stack objects and heap objects reallocated and safely managed in a protected memory area, where the memory vulnerabilities can be prevented with high probability. Once closing the green zone, OPSafe clears away all objects in the protected area and then frees the protected area. We have developed a prototype system and evaluated the system's effectiveness using four applications with a wide range of memory vulnerabilities. The experimental results demonstrate that our system can conveniently create and destruct a hot-portable green zone where the vulnerable application can survive crashes and eliminate erroneous execution.

Acknowledgment. This paper is supported by National High-tech R&D Program (863 Program) under grant No. 2012AA012600.

References

1. Berger, E., Zorn, B.: DieHard: Probabilistic Memory Safety for Unsafe Languages. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 158–168. ACM (2006)
2. Bhatkar, S., Sekar, R., DuVarney, D.: Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In: Proceedings of the 14th Conference on USENIX Security Symposium, pp. 271–286. USENIX (2005)
3. Chen, G., Jin, H., Zou, D., Zhou, B., Qiang, W., Hu, G.: SHelp: Automatic Self-healing for Multiple Application Instances in a Virtual Machine Environment. In: Proceedings of the 2010 IEEE International Conference on Cluster Computing, pp. 97–106. IEEE (2010)
4. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In: Proceedings of the 7th Conference on USENIX Security Symposium, pp. 63–78. USENIX (1998)
5. Gao, Q., Zhang, W., Tang, Y., Qin, F.: First-Aid: Surviving and Preventing Memory Management Bugs During Production Runs. In: Proceedings of the 4th ACM European Conference on Computer Systems, pp. 159–172. ACM (2009)
6. Prasad, M., Chiueh, T.: A Binary Rewriting Defense Against Stack Based Buffer Overflow Attacks. In: Proceedings of the 2003 USENIX Annual Technical Conference, pp. 211–224. USENIX (2003)
7. Qin, F., Tucek, J., Sundaresan, J., Zhou, Y.: Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. In: Proceedings of the 20th ACM Symposium on Operating System Principles, pp. 235–248. ACM (2005)
8. Rinard, M., Cadar, C., Dumitran, D., Roy, D., Leu, T., Beebee Jr., W.: Enhancing Server Availability and Security Through Failure-Oblivious Computing. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation, pp. 303–316. USENIX (2004)
9. Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., Keromytis, A.: ASSURE: Automatic Software Self-healing Using REscue points. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 37–48. ACM (2009)
10. Vision Solutions Staff, Assessing the Financial Impact of Downtime. Vision Solutions, Inc. (2010)
11. Chen, G., Jin, H., Zou, D., Zhou, B., Liang, Z., Zheng, W., Shi, X.: SafeStack: Automatically Patching Stack-based Buffer Overflow Bugs. To be appeared in IEEE Transactions on Dependable and Secure Computing. IEEE (2013)
12. Zou, D., Zheng, W., Jiang, W., Jin, H., Chen, G.: Memshepherd: Comprehensive Memory Bug Fault-Tolerance System. To be appeared in Security and Communication Networks. John Wiley & Sons, Ltd. (2013)
13. Avijit, K., Gupta, P., Gupta, D.: TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection. In: Proceedings of the 13th USENIX Security Symposium, pp. 45–56. USENIX (2004)