A Lightweight Buffer Overflow Protection Mechanism with Failure-Oblivious Capability

Tz-Rung Lee[1], Kwo-Cheng Chiu[1], and Da-Wei Chang[2]

[1] Department of Computer Science, National Chiao-Tung University, Hsin-Chu, Taiwan
`roylee17@gmail.com, inaba178@gmail.com`
[2] Department of Computer Science and Information Engineering, National Cheng-Kung University, Tainan, Taiwan
`davidchang@csie.ncku.edu.tw`

Abstract. Buffer overflow has become a major source of network security vulnerability. Traditional schemes for detecting buffer overflow attacks usually terminate the attacked service, degrading the service availability. In this paper, we propose a lightweight buffer overflow protection mechanism that allows continued network service. The proposed mechanism allows a service program to reconfigure itself to identify and protect the vulnerable functions upon buffer overflow attacks. Protecting only the vulnerable functions, instead of the whole program, keeps the runtime overhead small. Moreover, the mechanism adopts the idea of failure-oblivious computing to allow service programs to execute through memory errors caused by the attacks once the vulnerable functions have been identified, eliminating the need of restarting the service program upon further attacks to the vulnerable functions. We have applied the mechanism on five Internet servers. The experiment results show that the mechanism has little impact on the runtime performance.

Keywords: Buffer Overflow Attacks, Network Security, Self Reconfiguration, Failure-Oblivious Computing, Guard Pages.

1 Introduction

Buffer overflow has become a major source of network security vulnerability. Attackers can exploit this vulnerability by overwriting critical control data with maliciously crafted content so as to gain the full control of the program. According to the statistics from Common Vulnerabilities and Exposures (CVE), more than half of the software vulnerabilities come from buffer overflow since 2002, and the number of buffer overflow vulnerabilities is still growing. Numerous approaches have been proposed to detect buffer overflow attacks. Static analysis techniques [1,2] analyze the source code of the service programs to detect memory error problems while dynamic techniques [3-13] check data integrity during program execution. Although these techniques effectively detect attacks, they can not protect the processes from being compromised, and thus terminating the compromised processes is necessary to prevent further error propagation. In recent years, several techniques have been proposed to recover the attacked services, instead of terminating them, so as to provide continued service under the attacks [14-20]. However, such techniques cause moderate to substantial impact on the system performance.

In this paper, we propose a lightweight protection mechanism that provides continued service under buffer overflow attacks. The proposed mechanism allows a service program to reconfigure itself to collect runtime information in the face of buffer overflow attacks, to identify the vulnerable functions, and finally to protect those functions. Moreover, the mechanism adopts the idea of failure-oblivious computing [15,16] to allow service programs to execute through memory errors without losing their correctness once the vulnerable functions have been identified. This eliminates the need of restarting the service program upon further attacks so that continued service can be provided.

The mechanism adopts a multi-stage approach. At the initial stage, the service program is protected by a lightweight protection mechanism. Upon detecting an attack, the program performs stage transition and reconfigures itself to collect runtime information so as to identify the vulnerable functions. Once the vulnerable functions have been identified, state transition and self-reconfiguration are done again for applying more advanced (but more heavyweight) protection techniques on the functions, allowing the program to execute through further attacks. Applying more advanced protection techniques only on the vulnerable functions, instead of the whole program, helps to keep the runtime overhead small, and thus increases the feasibility of the mechanism.

In the current implementation, we utilize Address Space Layout Randomization (ASLR) [10] and Guard Page [17,18] as the aforementioned lightweight and heavyweight protection techniques, respectively. We have applied the mechanism on five Internet servers. The experimental results indicate that, in most of the cases, the proposed mechanism incurs less than 5% of runtime overhead.

The rest of this paper is organized as follows. Section 2 describes the related work. More detailed description about the ASLR and the Guard Page protection techniques are given in Sect. 3. Section 4 presents the design and implementation of our mechanism. Section 5 shows the experiment results, which are followed by the conclusions in Sect. 6.

2 Related Work

Originally, most of the related research efforts were put on the detection, either statically or dynamically, of the attacks. In recent years, more research efforts focus on recovering the attacked programs. In this section, we will describe those efforts.

2.1 Buffer Overflow Detection

Static buffer overflow detection techniques [1,2] analyze program source code to detect memory errors. The major limitation of such techniques is the lack of program runtime information. Numerous dynamic techniques have been proposed to detect buffer overflows during program execution. Several techniques detect buffer overflow attacks by checking the integrity of control data (i.e., return address, frame pointers, etc) [3,11]. StackGuard [6,7] and ProPolice [9] place canary values between local buffers and control data on stacks to check if the control data was corrupted due to buffer overflow. StackSheild [13] and RAD [4] copy the return address into a global return stack so that they can check the integrity of the return address in the function epilog. PointGuard [5] augmented the GCC compiler to emit code that encrypts and decrypts pointers before and after they are stored in memory, respectively, so as to detect invalid pointer updates. Address Space Layout Randomization (ASLR) [10] shifts memory segments (e.g. stack, heap, and shared library code) in the process address space with random offsets to obscure the target addresses from the attackers. Non-executable buffers [8,10,12] prevent execution of code on stacks or heaps with hardware assistance. An attacker may still inject instructions into the buffers, but any attempt to execute those instructions will cause an exception. Due to the limitation of the static techniques, some systems such as Cyclone [29] and CCured [30] combine static analysis and runtime checks. They statically check the source code for buffer overflow problems, and insert runtime checks for those which can not be identified statically.

2.2 Automatic Recovery from Attacks

Instead of terminating the victim service upon detecting an attack, several techniques have been proposed to recover the service so as to allow continued service. DIRA [20] augmented the GCC compiler to log memory updates and track data dependency during the program execution. When an attack is detected, the program identifies the external input data that corresponds to the attack and passes the data as a signature to the front-end filter. Finally, the program is rolled back to the state before the reception of the attack request. This technique degrades the runtime performance due to the overhead of logging memory updates and tracking data dependency. TaintCheck [14] executes programs in an emulation environment, which tags data derived from untrusted sources, such as network, as tainted and tracks its propagation in the program memory. Any attempt to use the tainted data as a pointer will be recognized as an attack and triggers the post-analysis procedure to provide information to the filter. The use of the emulation environment has a substantial impact on the program performance. According to the performance results on an Apache server, the measured slowdown ranges from about 2 to 25 times, which limits the practical usage. Sidiroglou and Keromytis [17,18,19] treated each execution of functions as a transaction. Once an error is detected, they rollback the memory changes caused by the function, abort the function, and continue the execution from where the function returns. However, they can not roll back I/O operations, and hence the program may not work in a consistent way in its continued execution. Rinard et al. [15,16] proposed the concept of failure-oblivious computing, which allows a program to execute through memory errors without compromising its correctness. They modified the CRED safe-C compiler [11] to augment the generated code to perform bounds checks and to store away or discard out of bounds writes. As a result, no memory data can be clobbered by buffer overflow. However, bound checks degrade the runtime performance (e.g. the measured slowdown ranges from 3% to 8.9 times) and make the approach less attractive for many applications.

In this paper, we propose a lightweight buffer overflow protection mechanism, which adopts the idea of failure-oblivious computing. Unlike the existing work, our self-reconfigurable and multi-stage design allows the program to achieve the same goal with little performance overhead.

3 Background

In this section, we briefly introduce the two existing techniques in our buffer overflow protection mechanism: Address Space Layout Randomization (ASLR) and Guard Page.

ASLR shifts memory segments (e.g. stack, heap, and shared library code) in the process address space with random offsets to obscure the target addresses from the attackers. As illustrated in Fig. 1, attackers have to guess the target address, and a wrong guess usually leads to a segmentation fault. The limitation of ASLR is that it can be defected by brute-force guessing [21]. Despite the limitation, ASLR has been integrated in many systems [22,23] due to its ability to detect a broad range of memory errors and negligible runtime overhead.

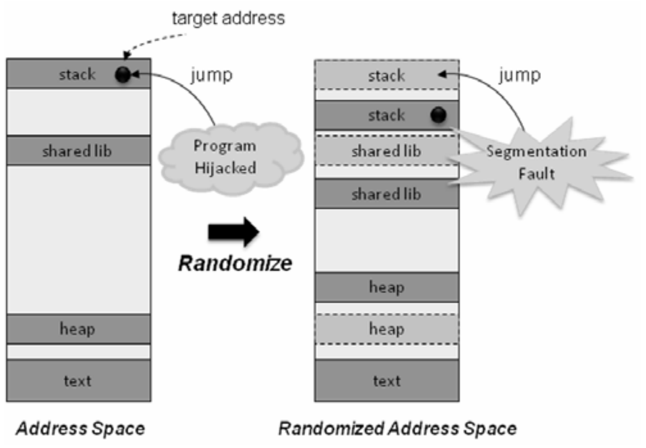

Fig. 1. Address space layout randomization

The Guard Page mechanism allocates extra inaccessible memory pages (i.e., guard pages) adjacent to the allocated memory areas during memory allocations so as to detect memory overflow. This mechanism has been used in debugging errors of heap based memory for many years [25], and it can also be applied for detecting stack-based buffer overflow vulnerability [17,18] by repositioning the buffer to the heap area [24] and guarding it with a guard page, as shown in Fig. 2. Any attempt to overflow the guarded buffers causes a segmentation fault and thus reveals the attack. Therefore, guard pages can detect attacks before the memory data adjacent to the guarded buffers has been compromised. Based on this property, we are able to allow the program to execute through buffer overflow attacks, realizing the concept of failure-oblivious computing. The drawback of guard pages is its high runtime overhead. Each buffer allocation and deallocation requires additional system calls for mapping and unmapping guard pages. Thus, it is impractical to guard all the buffers in a

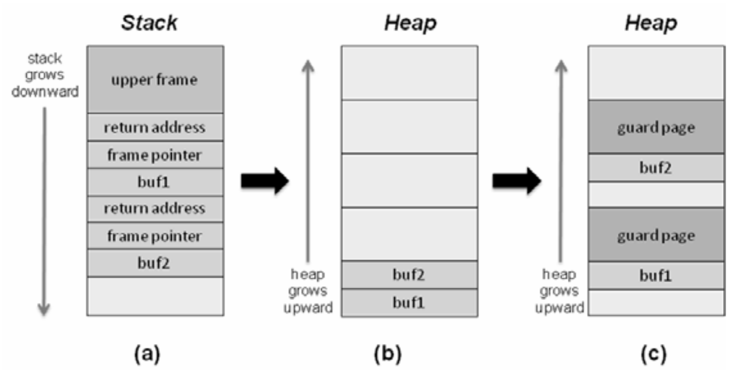

Fig. 2. Repositioning buffers from stack to heap and then guard them with guard pages

program. Our protection mechanism uses runtime information to reduce the number of buffers that need to be guarded, and thus leads to little performance impact.

4 Design and Implementation

In this section, we present the design and implementation of the proposed buffer overflow protection mechanism.

4.1 Function-Based Protection

We focus on the stack-based buffer overflow attacks, so only local buffers are concerned in the current implementation. We treat functions that have local buffers as potentially vulnerable functions, and we apply protection techniques in a function-by-function basis. Each potentially vulnerable function has two versions: the O_VERSION and the G_VERSION. The former whose function name is prefixed with O is the original version, and the latter whose function name is prefixed with G uses guard pages to protect its local buffers. Both versions are generated by using source code transformation. Figure 3 illustrates an example of the transformation. As shown in Fig. 3(a), two functions, B and D, are potentially vulnerable. After transformation, both B and D are transformed into two versions of function implementations, as shown in Fig. 3(b). In the figure, O_B and O_D are the original functions, while G_B and G_D are functions that allocate their local buffers from the heap and protect the buffers with guard pages. Note that the functions with original names, such as B and D, are transformed into wrapper functions, which invoke one version of the function implementations based on the decision of a proxy function. Specifically, each wrapper function invokes the proxy function with the unique identifier of the former as the parameter, and the proxy function returns either O_VERSION or G_VERSION so that the wrapper function can then invoke the function implementations accordingly. The unique function identifiers are assigned during the transformation process.

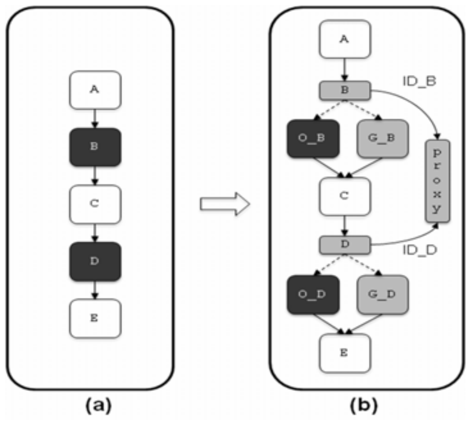

Fig. 3. Transformation of potentially vulnerable functions

4.2 Multi-stage Self-reconfiguration

The proposed mechanism adopts a multi-stage approach. Figure 4 shows the stages and the transitions among the stages. The default stage, which is the initial stage, aims to provide effective attack detection without degrading the service performance. In this stage, ASLR is applied and the original function implementations are always chosen. Once an attack is detected, the program transits to the logging stage, which uses a lightweight technique to collect runtime call stack information. The information is collected by pushing the function identifier into a separated and protected stack right before the invocation of each transformed function (i.e., potentially vulnerable function) and popping the function identifier out of the stack on the return of the invocation. On detecting an attack in this stage, the program passes the call stack information, which is referred to as the candidate list in the rest of the paper, to the watching stage and then transits to that stage. Similar to the default stage, the logging stage also relies on the ASLR as the attack detection technique and the original function implementations are always chosen.

Given the candidate list, the watching stage can use the Guard Page mechanism to protect buffers in the functions that appear in the candidate list. This is achieved by returning G_VERSION when the proxy function is invoked by a function whose identifier appears in the candidate list. Once a further attack arrives and causes the overflow of a guarded buffer, the corresponding vulnerable function can be identified. In that situation, the program exports the identifier of the vulnerable function and transits to the protecting stage, which protects only the buffers allocated in that vulnerable function and allows the program to execute through further attacks without losing its correctness. In the protecting stage, the proxy function returns G_VERSION only when it is invoked by the wrapper of the vulnerable function. Protecting only a small number of buffers leads to little performance overhead. Note that, all stages use ASLR as a program-wide detection technique, which means that the buffers protected with guard pages are still protected with ASLR as well. However, overflowing such buffers is always detected first by the guard page.

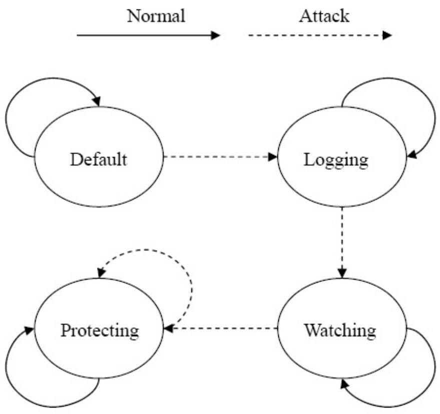

Fig. 4. Stage transition diagram

Instead of including the control-path determination logic for all the stages into a single proxy implementation, we choose to have multiple per-stage proxy implementations. Each proxy implementation is realized as a single shared object. During each stage transition, the program reconfigures itself by linking to a different proxy implementation. Files are used for information passing between proxy implementations.

5 Performance Evaluation

In this section, we present the performance results of the proposed buffer overflow protection mechanism. We have applied the protection mechanism on five open source network server programs, as shown in Table 1. Source code transformation is done by using TXL [26], a special-purpose programming language designed for software analysis and source code transformation. We extend the Gemini tool [24], which transforms source code to reposition stack-based buffers on the heap by using TXL, to protect local buffers with guard pages. We compare the performance of the original and transformed versions of each test program. For the transformed version, we send attack messages from the client machine to trigger program reconfigurations and stage transitions, and run benchmarks to measure the performance of each stage.

The experimental environment consists of a server machine and a client machine, which are connected via a 100MB/s Ethernet switch. The server machine is equipped with a 2.0 GHz Pentium 4 processor and 512MB RAM while the client machine is

Table 1. List of the vulnerable programs

Vul. Programs	Description	Vulnerability
Qpopper 4.0.4	POP3 Mail Server	CVE-2003-0143
dproxy-nexgen	Caching DNS Server	CVE-2007-1866
ProFTPD 1.3.0a	FTP Server	CVE-2006-6563
ghttpd 1.4-3	Web Server	CVE-2002-1904
Apache (with mod_jk 1.2.0)	Web Server	CVE-2007-0774

equipped with a 3.4 GHz Pentium 4 processor and 768MB RAM. Both machines run Linux Kernel 2.6.21.

Figure 5 shows the results on Qpopper, a popular POP3 mail server. The vulnerability is due to a call to Qvsnprintf() within pop_msg() in popper/pop_msg.c, which leaves a buffer non-terminated and can be exploited to execute arbitrary code via a buffer overflow. We use Postal [27], a benchmark for measuring performance of SMTP and POP3 servers, to create POP3 sessions in a saturated manner on the server, which has 2000 mailboxes with a total size of 50M bytes of messages. A typical POP3 session includes logging on the server, listing mail messages, retrieving messages and deleting messages. As shown in the figure, the performance degradation of the transformed Qpopper is little, ranging from 1% to 3%.

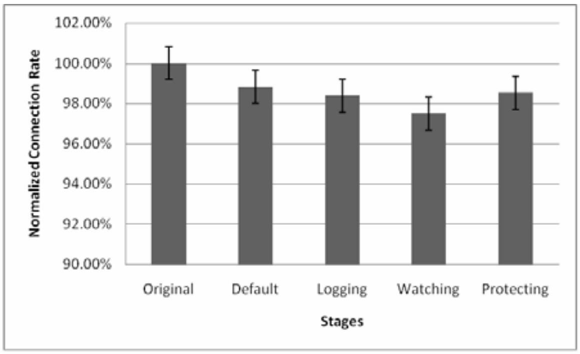

Fig. 5. Performance results of Qpopper

Figure 6 shows the results on Dproxy-nextgen, a small caching domain name server. It has a buffer overflow vulnerability within its dns_decode_reverse_name() function, which allows remote attackers to execute arbitrary code by sending a crafted packet to UDP port 53. In this experiment, we measure the response time of looking up a domain name. The figure indicates that all the stages of the transformed version have very low (below 2%) performance overhead.

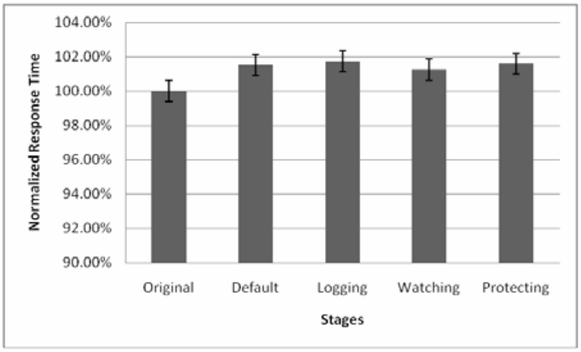

Fig. 6. Performance results of Dproxy-nexgen

Figure 7 shows the results on ProFTPD, a high-performance and highly configurable FTP server. The vulnerability is a boundary error within the pr_ctrls_recv_request() function in the src/ctrls.c file, which can be exploited to execute arbitrary code. We measure the performance by fetching files with different sizes from the server. As shown in the figure, the performance overhead of the watching stage can be larger than 10% when fetching small files. This overhead reduces as the file size grows. For example, there is no visible overhead when fetching files with 10M bytes. This is because IO transfer dominates the execution time in the cases of large file transfer. Note that, the transformed ProFTPD does not transit to the protecting stage in this experiment. This is because the overflows happen in the read() system call, and the Linux kernel handles these exceptions, which are triggered by guard pages, by doing early returns to the user space instead of issuing SIGSEGV signals to the program. As a result, no stage transition is triggered. In this case, the program memory is not compromised and the program can still continue its execution without losing its correctness.

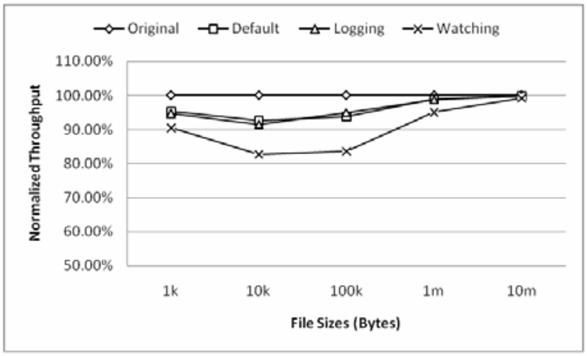

Fig. 7. Performance results of ProFTPD

Figure 8 shows the results on Ghttpd, a fast and efficient HTTP server with CGI support. The vulnerability is a buffer overflow within the Log() function in the util.c file, which allows remote attackers to execute arbitrary code via a long HTTP GET request. We use WebStone [28], a standard benchmark for web server, to evaluate performance of the transformed ghttpd. As shown in the figure, the performance overhead is less than 3% in the default, logging and protecting stages. Even for the watching stage, the overhead is still less than 5%.

Figure 9 shows the results on Apache Tomcat Connector (mod_jk), a module of Apache for connecting to Tomcat, which is a web container, or an application server that implements the servlet and the JavaServer Pages (JSP) specifications. The vulnerability is an unsafe memory copy within the map_uri_to_worker() function in the native/common/jk_uri_worker_map.c file, which can be exploited to execute arbitrary code or crash the web server by sending a long URL request. As the figure indicates, all the stages of the transformed Apache program have nearly the same performance with the original version.

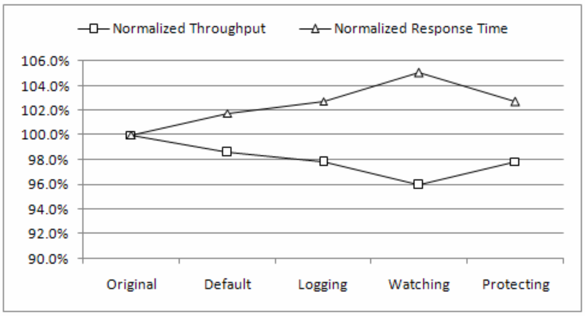

Fig. 8. Performance results of Ghttpd

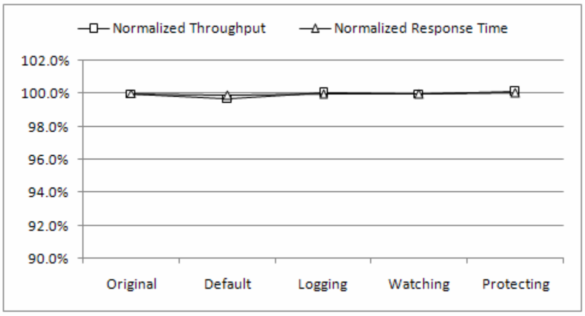

Fig. 9. Performance results of Apache

6 Conclusions

In this paper, we propose a lightweight protection mechanism that allows continued network service under buffer overflow attacks. Upon detecting a buffer overflow attack, the mechanism allows a service program to reconfigure itself to collect runtime call stack information, to locate the vulnerable functions, and finally to apply the Guard Page protection technique on those functions. Applying the Guard Page protection technique only on the vulnerable functions, instead of the whole program, keeps the runtime overhead small. Moreover, based on the concept of failure-oblivious computing, the proposed mechanism allows the service programs to execute through memory errors without losing their correctness once the vulnerable functions have been located, eliminating the need of restarting the service programs upon further attacks, and thus providing continued service.

In the current implementation, self-reconfiguration is achieved by source code transformation and dynamic linking. We have applied the mechanism on five Internet servers. According to the experiment results, the proposed mechanism incurs very little performance impact (i.e., less than 5% in most of the cases).

The original model mentioned in Fig. 4 can only deal with overflows of buffers in a single vulnerable function. However, it is possible that a network service has multiple vulnerable functions. To deal with this problem, we have extended the model. In the future, we will implement the extended model on network services that has multiple buffer overflow vulnerabilities and evaluate the performance of the model.

References

1. Dor, N., Rodeh, M., Sagiv, M.: Cssv: Towards a Realistic Tool for Statically Detecting all Buffer Overflows in C. In: ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 155–167 (2003)
2. Larochelle, D., Evans, D.: Statically Detecting Likely Buffer Overflow Vulnerabilities. In: 10th USENIX Security Symposium, pp. 177–190 (2001)
3. Baratloo, A., Singh, N., Tsai, T.: Transparent Run-time Defense against Stack Smashing Attacks. In: USENIX Annual Technical Conference, pp. 251–262 (2000)
4. Chiueh, T.C., Hsu, F.H.: RAD: A Compile-time Solution to Buffer Overflow Attacks. In: International Conference on Distributed Computing Systems, pp. 409–417 (2001)
5. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. In: USENIX Security Symposium, pp. 91–104 (2003)
6. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: 7th USENIX Security Conference, pp. 63–78 (1998)
7. Cowan, C., Wagle, P., Pu, C., Beattie, S., Walpole, J.: Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In: DARPA Information Survivability Conference and Exposition, pp. 119–129 (2000)
8. Dik, C.: Non-Executable Stack for Solaris, Posted to comp.security.unix (January 1997)
9. Etoh, H., Yoda, K.: Protecting from Stack-Smashing Attacks, `http://www.trl.ibm.com/projects/security/ssp`
10. The PaX Team: PaX Address Space Layout Randomization, `http://pax.grsecurity.net`
11. Ruwase, O., Lam, M.: A Practical Dynamic Buffer Overflow Detector. In: Network and Distributed System Buffer overflow Symposium, pp. 159–169 (2004)
12. Solar Designer: Non-Executable User Stack, `http://www.openwall.com/linux/`
13. StackShield, `http://www.angelfire.com/sk/stackshield`
14. Newsome, J., Song, D.: Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In: 12th Annual Network and Distributed System Security Symposium (2005)
15. Rinard, M., Cadar, C., Dumitran, D., Roy, D., Leu, T., Beebee, J.W.: Enhancing Server Availability and Security Through Failure-Oblivious Computing. In: 6th Symposium on Operating Systems Design and Implementation, p. 21 (2004)
16. Rinard, M., Cadar, C., Roy, D., Dumitran, D.: A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors). In: 20th Annual Computer Security Applications Conference, pp. 82–90 (2004)
17. Sidiroglou, S., Keromytis, A.D.: A Network Worm Vaccine Architecture. In: 12th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp. 220–225 (2003)
18. Sidiroglou, S., Keromytis, A.D.: A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. In: 8th Information Security Conference, pp. 1–15 (2005)
19. Sidiroglou, S., Locasto, M.E., Boyd, S.W., Keromytis, A.D.: Building a Reactive Immune System for Software Services. In: USENIX Annual Technical Conference, pp. 149–161 (2005)
20. Smirnov, A., Chiueh, T.C.: DIRA: Automatic Detection, Identification and Repair of Control-Hijacking Attacks. In: 12th Annual Network and Distributed System Security Symposium (2005)

21. Shacham, H., Page, M., Pfa, B., Goh, E.J., Modadugu, N., Boneh, D.: On the Effectiveness of Address-Space Randomization. In: 11th ACM Conference on Computer and Communications Security, pp. 298–307 (2004)
22. Liang, Z., Sekar, R.: Automated, Sub-Second Attack Signature Generation: A Basis for Building Self-Protecting Servers. In: 12th ACM Conference on Computer and Communications Security (2005)
23. Liang, Z., Sekar, R.: Automatic Generation of Buffer Overflow Attack Signatures: An Approach Based on Program Behavior Models. In: 21st Annual Computer Security Applications Conference, pp. 215–224 (2005)
24. Dahn, C., Mancoridis, S.: Using Program Transformation to Secure C Programs against Buffer Overflows. In: 10th Working Conference on Reverse Engineering, pp. 323–332 (2003)
25. Perens, B.: Electric Fence, `http://perens.com/FreeSoftware/ElectricFence`
26. Cordy, J.R., Dean, T.R., Malton, A.J., Schneider, K.A.: Source Transformation in Software Engineering using the TXL Transformation System. Journal of Information and Software Technology 44(13), 827–837 (2002)
27. Coker, R.: Postal Benchmark, `http://www.coker.com.au/postal`
28. Mindcraft Inc.: WebStone: the Benchmark for Web Servers, `http://www.mindcraft.com/benchmarks/webstone`
29. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: a Safe Dialect of C. In: USENIX Annual Technical Conference, pp. 275–288 (2002)
30. Necula, G.C., McPeak, S., Weimer, W.: CCured: Type-Safe Retrofitting of Legacy Code. In: 29th ACM Symposium on Principles of Programming Languages, pp. 128–139 (2002)