國 立 交 通 大 學

多媒體工程研究所

碩 士 論 文

具有錯誤忽略能力之輕量級緩衝溢位保護機制

A Lightweight Buffer Overflow Protection Mechanism with Failure-Oblivious Capability

研 究 生：李子榮

指導教授：張瑞川 教授

中 華 民 國 九 十 六 年 六 月

具有錯誤忽略能力之輕量級緩衝溢位保護機制

研究生：李子榮　　　　　　　　　　指導教授：張瑞川教授

國立交通大學多媒體工程研究所

摘 要

緩衝溢位漏洞一直是個相當重要之網路安全議題。過去以來，多數的防治機制大多著重在於攻擊的偵測方面。為了降低受到攻擊所造成的傷害，這些防治機制在偵測到攻擊時，便終止受到攻擊之程式或是在有需要的情形下，重新啟動該程式。然而在面對自動化的重複攻擊時，這樣的作法對於多數的網路伺服器而言，卻不是一個理想的解決方式，因為不斷地重新啟動程式將大幅降低程式所能提供的服務能力。近年來，部份研究著重在將程式從受到攻擊後的狀態中回復並繼續執行。雖然這些機制能夠使程式能在自動化的重複攻擊之下，還能保持一定的服務能力，但是也同時衝擊了程式在平時效能。

這此論文中，我們提出了一個輕量級的機制。在攻擊回復的方面，應用了錯誤忽略的概念。這個機制透過將原始碼做轉換，同時對程式應用了多個保護技術。經過轉換的程式，可利用輕量級的技術提拱整體性的保護，並利用在執行期所蒐集的資訊，以函式為單位，選擇性針對部份的程式使用較重量級的技術，加強弱點的防護。我們的實驗數據顯示，轉換過之 Apache 伺服器只造成了極小的效能負擔，卻能在面對自動化的重複攻擊時，維持 60%到 70%的服務能力。而未經保護之版本，在同樣的攻擊下，只能提供低於 10%的服務能力。

Student: Tz-Rung Lee **Advisor: Dr. Ruei-Chuan Chang**

Computer Science and Engineering, College of Computer Science

National Chiao Tung University

Abstract

Buffer overflow vulnerability is a severe security problem due to insufficient bound checking of programs. Most research efforts were put on the detection of the attacks. Many proposed techniques terminate the compromised process upon detecting an attack and restart a new instance if necessary. However, while facing automated and repetitive attacks, terminating the compromised instance and restart a new one is probably not a desired reaction for most network services since it degrades the service availability. In the recent years, more research efforts focused on preserving service availability under repetitive attacks. However, while preserving service availability, their mechanisms also have a substantial impact on the performance of protected programs.

In this paper, we propose a lightweight mechanism which adopts the idea of failure-oblivious computing on recovering programs from buffer overflow attacks. The proposed mechanism automatically transforms a program to apply multiple protection techniques on the program in a function-by-function basis. The transformed program minimizes performance overhead by selectively enabling heavyweight protection for only a small set of functions according to the run-time information collected during its execution. Our experiment results indicate that all transformed programs have very low impact on performance. It also indicates that the transformed Apache server preserves from 60% to 70% of service availability while the unprotected version renders less than 10% of service.

Table of Contents

List of Figures

1 Introduction

Buffer overflow vulnerability is a common programming error due to insufficient bound checking. Users can overflow a buffer and hence pollute the data adjacent to the buffer. Attackers exploit this by supplying specially designed data to overwrite the memory data and alter the program control flow to cause it crash, or even worse, gain the fully control of the program.

1.1 Motivations

Over the past years, buffer overflow has become the major source of network security vulnerability. According to the statistics from Common Vulnerabilities and Exposures (CVE), which is showed in Table 1, more than half of the software vulnerabilities come from buffer overflow since 2002. As the software gets more complex, the probability of producing bugs also gets higher.

Table 1. Statistics of buffer overflow vulnerability from CVE

Year	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006
# of Vulns	104	107	294	372	815	985	697	1389	3224	4765
% of Total	41%	43%	32%	37%	49%	50%	54%	59%	66%	72%

Most of the research efforts were put on the detection of the attacks. Static analysis techniques such as [9] [13] detect memory error problems at source code level while dynamic techniques check data integrity on control flow to ensure the execution is not compromised. Although these techniques effectively detect attacks, they can not protect the process itself from being compromised and terminating the compromised process is necessary to cease further error propagation.

With repetitive attacks, however, terminating the compromised process and restarting a new instance degrade the service availability. In the recent years, automated attacks which try to exploit the buffer overflow vulnerability in faster and repetitive ways have motivated the research on recovering the program from attacks to preserve service availability under such attacks. DIRA [27] saved program memory state during runtime execution and recovers it after attacks. However, only rolling back program memory can not eliminate side effects such as modification to file system. TaintCheck [18] run program on an emulator for monitoring program execution. Rinard et al. [20] [21] 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 [22] so that instead of terminating the process upon detecting a buffer overflow, they discard the out of bounds writes and continue the execution. All these techniques tried to preserve service availability under repetitive attacks at a cost of runtime performance of the protected programs.

1.2 Our Contributions

We proposed a lightweight multi-stage design buffer overflow protection mechanism, which recovers programs from attacks and has little impact on the performance of protected programs. The idea is to allow the protected program to collect runtime information during execution in earlier stages, and then pass the information to later stages so that only the vulnerable part of the program is protected thus minimizes the runtime overhead.

We include several open source network servers in the experiments to evaluate the performance overhead and the improvement of service availability under repetitive attacks. The experiment result indicate that all the transformed programs have less than 3% overhead at the initial stage and it also shows that our mechanism improves the service availability under repetitive attacks. The protected version of Apache server preserves from 60% to 70% of service availability while the unprotected version renders less than 10% service.

1.3 Our Approach

To realize the aforementioned multi-stage mechanism, we transform the program so that each potentially vulnerable function is transformed to two versions: original and protected versions, with a proxy function that decides which one is executed in during runtime execution. The proxy function together with other helper functions is actually an interface, which can be implemented and dynamically linked to the transformed program. In this way, we can have several different implementations, which we call them switches in this paper, and change the behavior of the transformed program by linking different switch to it. In the context of rest of this paper, running at a specific stage means that the program is linked to a switch designed for that stage.

To recovery programs from attacks, we transform program source code to reposition stack-based buffers to the heap and protect them with read-only guard pages. In this way, overflowing such buffers will cause a segmentation fault and be detected. Since the overwritten does not clobber the program memory, the program can execute through the buffer overflow without losing correctness, which shares the same idea with failure-oblivious computing [20] [21].

1.4 Organization of the Paper

The rest of this paper is organized as follows. In Chapter 2, we discuss two existing techniques used in our mechanism. In Chapter 3, we discuss the design and implementation of our mechanism. We present experiment results and evaluation in chapter 4. We review several other techniques to address buffer overflow problem in Chapter 5 and conclude this paper in Chapter 6.

2 Background

As mentioned before, the proposed multi-stage buffer overflow protection mechanism applies different protection techniques on different parts of a program. In this chapter, we introduce two existing techniques that we used in our protection mechanism: Address Space Layout Randomization (ASLR) and Guard Pages.

2.1 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) [19] 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 Figure 1, attackers have to guess the target address, and a wrong guess usually leads to a crash due to a segmentation fault, which can be easily detected.

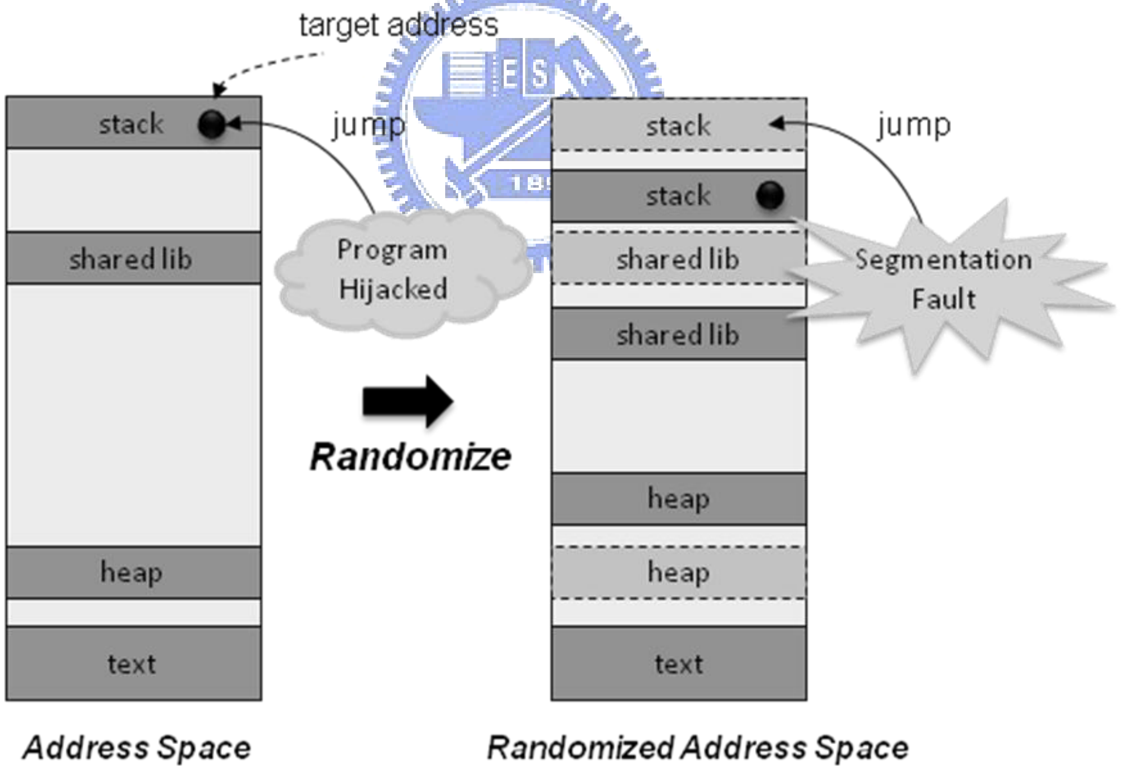

Figure 1. Address Space Layout Randomization

The effectiveness of ASLR relies on the low chance for attackers to guess the right target address. However, attackers may still defeat ASLR by brute-force guessing [23]. Despite the limitation, ASLR has been integrated in many systems due to its negligible runtime overhead.

It is also integrated to detect buffer overflow attacks in some security systems [14] [15] since it detects a broad range of memory errors.

2.2 Protecting Local Buffers with Guard Page

To obstruct stack-based buffer overflow attacks, Gemini [8] transforms programs to reposition stack-allocated buffers to the heap at compile time. Figure 2 shows a simplified example of such transformation. Generally speaking, heap-based buffer overflow vulnerabilities are much hard to be exploited than stack-based ones. However, attackers still have chances to compromise the program [12]. Besides, the transformed program can not prevent data from being compromised and hence can not execute through the attacks.

```
int func() {
    char buf[1024];
    ...
    other_func(buf);
    ...
    return 0;
}
```

```
int func(){
    char *buf = malloc(1024);
    ...
    other_func(buf);
    ...
    free(buf);
    return 0;
}
```

Figure 2. Reposition Buffers from Stack to Heap with Source Code Transformation

The use of guard pages can solve above problems. Mapping additional inaccessible memory region during memory allocation operations is a common debugging technique. This region may contain one or more pages, which are referred as guard pages. Guard pages have been used in debugging errors of heap allocated memory for many years [2]. Recently, it is also found in some systems, such as Linux kernel, for detecting stack overflow of kernel space.

Combining with the transformation described earlier, the same technique can be also applied to detecting stack-based buffer overflow vulnerability [24] [25]. Figure 3 illustrates an example of such combination. Any attempt to overwrite memory adjacent to the guarded buffers causes a segmentation fault and thus reveals the attack. In other words, guard pages can detect attacks before memory data are 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 allocation and deallocation of buffers requires additional system calls for mapping and unmapping guard pages. Thus, it is impractical to guard all the buffers in a program. Our protection mechanism uses runtime program information to reduce the number of buffers to be guarded, and thus leads to little performance impact, as shown in the experimental results.

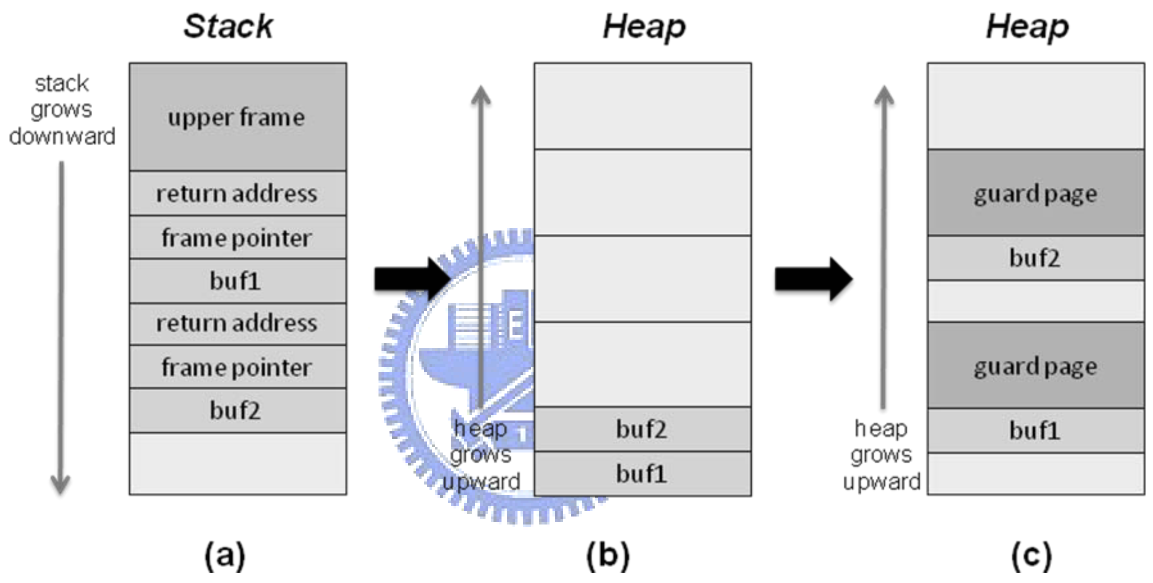

Figure 3. Reposition Buffers from Stack to Heap and Guard Them with Guard Pages

3 Design and Implementation

In this chapter, we present a multi-stage lightweight buffer overflow protection mechanism that has the following features. First of all, by collecting runtime process information, it only needs to protect buffers in the vulnerable functions, minimizing the runtime overhead. Second, it takes advantage of the idea of failure-oblivious computing to execute through buffer overflows.

The proposed mechanism consists of five stages, as shown in Figure 4. The *default* stage, which is the initial stage, aims to provide effective attack detection without degrading the program performance. We achieve this by applying ASLR on this stage. Once an attack is detected, the program transits to the *logging* stage, which uses a lightweight technique to collect run-time call stack information. On detecting an attack in this stage, the program passes that information to the *watching* stage and then transits to that stage. Similar to the *default* stage, the *logging* stage also relies on the ASLR as its attack detection technique.

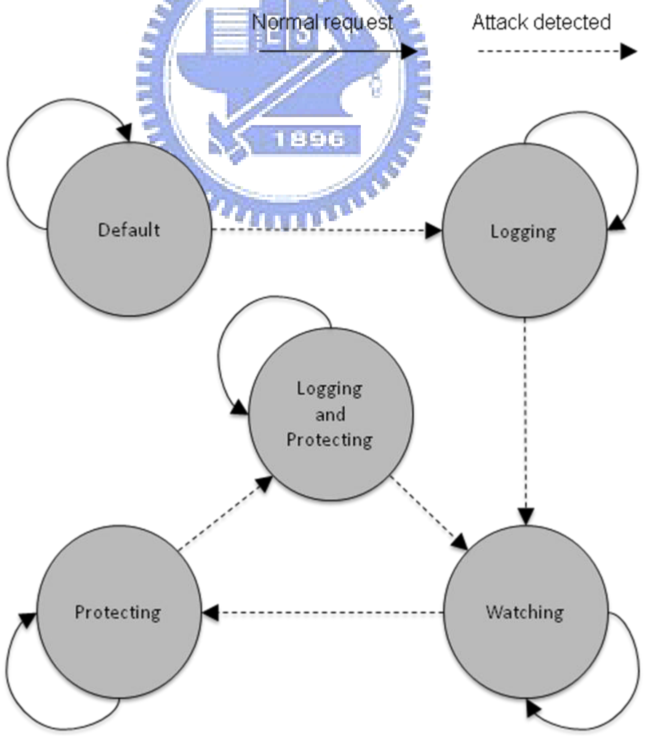

Figure 4. Stages and the Transition Diagram

With the run-time call stack information, the *watching* stage can apply protection mechanism on a relatively smaller number of buffers, compared to the number of buffers in the whole program. Specifically, it needs to protect only the buffers in the functions that appear in the call stack while the attack was detected. The *watching* stage protects each of the buffers by using the guard page mechanism, and hence the overflowed buffer can be identified once a further attack arrives. In that situation, the program transits to the *protecting* stage, which protect the buffers allocated in that single function (i.e., the vulnerable function) and allows the program to execute through further attacks without losing correctness. 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. If the program contains more than one vulnerable function, attacks to buffers residing in unidentified vulnerable functions (i.e. buffers that are not protected with guard pages) will be detected by ASLR and a similar process is repeated. In this case, the program transits to the *logging and protecting* stage, which collects run-time call stack information again while at the same time protects buffers in the previously identified vulnerable functions with guard page.

The proposed mechanism has little performance impacts on the program because that it applies heavyweight protection technique (that is, the guard page technique) on only a small number of buffers. For example, only the buffers allocated by the vulnerable function are protected in the *protecting* stage. Before these buffers are identified, merely lightweight protection mechanisms such as ASLR are applied.

To realize the aforementioned multi-stage protection mechanism, we should be able to apply multiple protection techniques concurrently on a program, and selectively enable different techniques on different parts of the program during its run-time execution. We achieve this by using the source code transformation and dynamic linking techniques, which will be described in Section 3.1. The mechanisms of stage transition and inter-stage communication will be presented in Section 3.2. Finally, the implementation details of each stage will be given in Section 3.3.

3.1 Function-based Protection

We treat functions that have local buffers as potentially vulnerable functions[1], and we apply protection techniques in a function-by-function basis. Each potentially vulnerable

[1] Currently, we focus on the stack-based buffer overflow attacks, so only local buffers are concerned in the current implementation.

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 5 illustrates an example of the transformation. As shown in Figure 5 (a), two functions, B and D, are potentially vulnerable. After transformation, both B and D are transformed into two versions, as shown in Figure 5 (b). Functions whose names are prefixed with O, such as O_B and O_D, are the original functions, and functions whose names are prefixed with G, such as G_B and G_D, are those that allocate 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 a system-wide proxy function to determine the control path. On invoking the proxy function, each wrapper function passes its function identifier so that the proxy function can determine the control path. Each function identifier is unique and assigned in the transformation process.

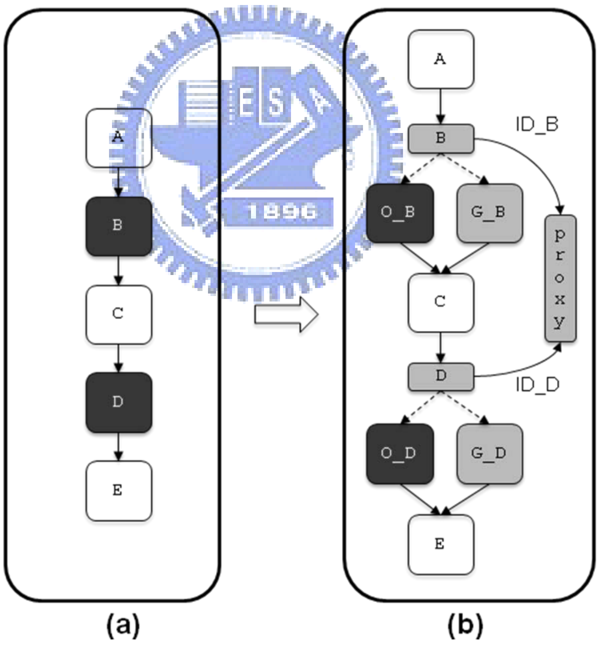

Figure 5. Illustration of the Transformation

A proxy function determines the control path based on the current stage and the information passed from the previous stage. For example, the *watching* stage guards the buffers belonging to the functions that were appeared in the call stack information, which was passed from the *logging* stage. 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 and switch these implementations during each stage transition. This maintains the extensibility of our multi-stage protection mechanism because that it is straightforward to insert/delete stages. To facilitate proxy implementation switching, each proxy implementation is realized as a single shared object which can be dynamically linked into the program.

Figure 6 illustrates an example of changing control path by switching proxy implementations. The program shown in Figure 6 (a) always executes the original versions of the potentially vulnerable functions, while the program shown in Figure 6 (b) always executes the versions with guard pages.

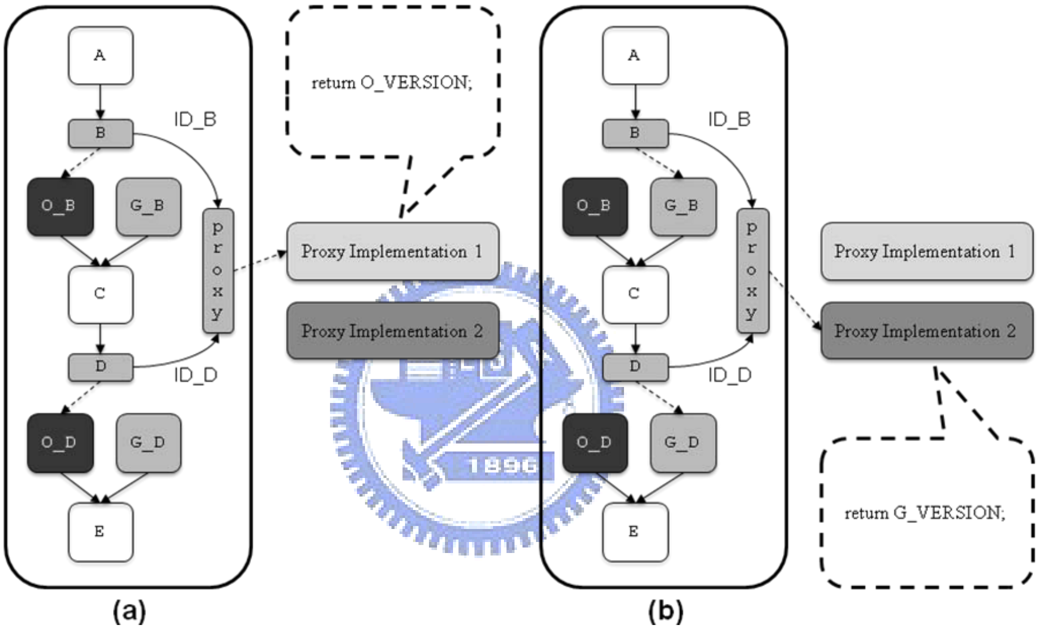

Figure 6. Change Program's Control Flow with Different Switches

Besides the proxy function, a proxy implementation also consists of some helper functions. For example, it consists of an initialization function, in which a proxy implementation can obtain the information passed from the previous stage. Moreover, it also contains a pair of prologue and epilogue functions, which are invoked by the wrapper function before and after the invocation of proxy function, respectively. With the prologue/epilogue functions, the *logging* stage can record the call stack information. Figure 7 shows the proxy interface, which consists of a set of the aforementioned four functions. Each proxy implementation switch changes the implementation of the proxy interface.

```
void __attribute__((constructor)) dynsw_init();
int dynsw_proxy( int func_id);
void dynsw_prologue( int func_id);
void dynsw_epilogue( int func_id);
```

Figure 7. The Proxy Interface

3.2 Stage Transition and Inter-Stage Communication

Stage transition is managed by a monitor process, which sets up the linking between the proxy implementation of the next stage and the transformed program, and then restarts the latter. As mentioned before, a proxy implementation is a shared object which can be dynamically linked with the program during the loading of the program. In a UNIX-like system, the link setup can easily be done by setting the LD_PRELOAD environment variable to the path of the proxy implementation before the program is loaded to run.

Figure 8 shows the interactions between the transformed program and the monitor program. As shown in the figure, the program has the ability to detect attacks since we apply detection techniques on it. Both ASLR and guard pages cause the program to trigger a segmentation fault signal (SIGSEGV) upon detecting an attack, and each proxy implementation registers a segmentation fault handler, which creates a file named DYNSW_RESTART to notify the monitor to start stage transition.

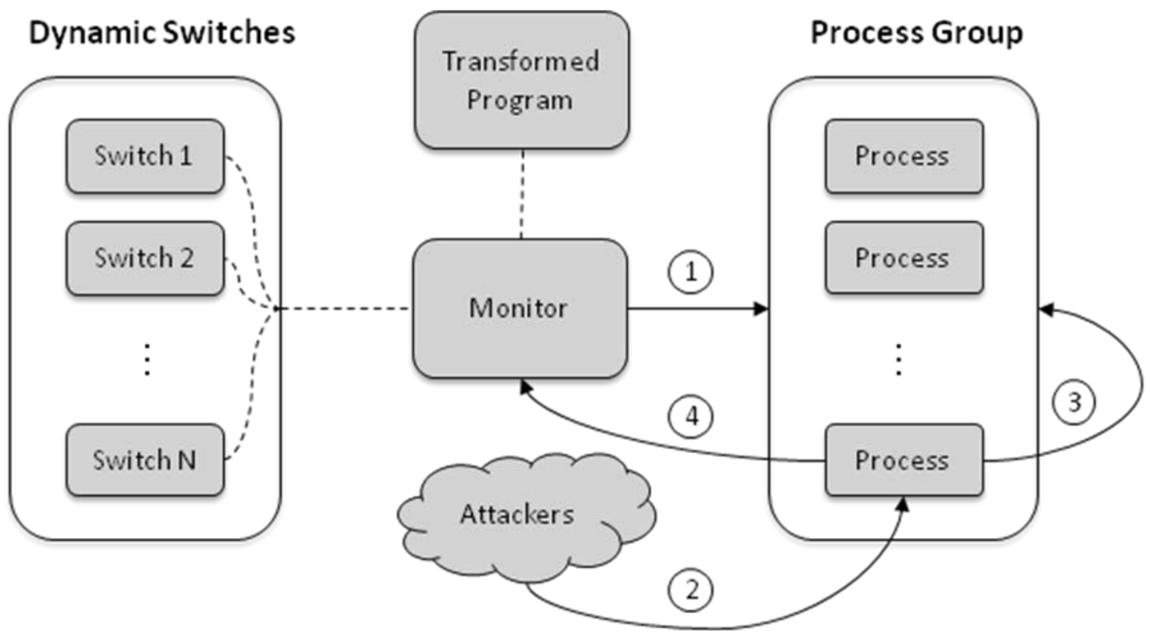

Figure 8. Interactions between the Transformed Program and the Monitor

For multi-process programs, it is necessary to terminate all running instances linked with the current proxy implementation before transiting to the next stage. It is possible to achieve this by sending a SIGABRT signal to those processes through the monitor. However, this requires the monitor to track processes relationship (e.g. process group and process session) and difficulties can arise if processes, especially daemons, change the groups of their descendent processes during execution. Therefore, we achieve this by sending a SIGABRT signal from the attacked process to abort other process in the same process group before it is terminated. Figure 9 shows the routine we use to do the stage transition in the SIGSEGV signal handler.

```
INLINE void next_state(){
        /* notify the monitor to transit to the next stage */
        creat( DYNSW_RESTART, 00600);
        /* terminate all instances that are linked with current proxy implementation */
        kill( 0, SIGABRT);
}
```

Figure 9. Routine Called in SIGSEGV Signal Handler for Stage Transition

Some programs register their own SIGSEGV signal handler. Since our SIGSEGV handler is registered during the program initialization, any later registration will cause our registration in vain. We solve this problem by commenting the code related to signal handler registration in the program transformation phase.

As mentioned before, a proxy implementation can pass the information it has collected to the next stage. Since all the processes of a program run at current stage are terminated before stage transition, the information must be saved outside the process address spaces. For simplicity, we use files for such information passing. A program exports information to a file before it is terminated, and the restarted instance (i.e., the next stage) imports that information during its initialization.

3.3 Multi-Stage Buffer Overflow Protection Mechanism

In this section, we present the implementation details of each stage in our multi-stage protection mechanism. The proxy implementation of the *default* stage is shown in Figure 10. The dynsw_proxy() function simply returns O_VERSION to tell the wrapper function to choose the original version. Both dynsw_prologue() and dynsw_epilogue() does nothing since there is no need to record information during function entry and exit in this stage.

```
int dynsw_proxy( int func_id){ return O_VERSION; }
void dynsw_prologue( int func_id){ /* do nothing */ }
void dynsw_epilogue( int func_id){ /* do nothing */ }
INLINE void segv_handler( int signum, siginfo_t *siginfo, void *ucontext){
    next_state();
}
```

Figure 10. *Default* Stage Implementation

The proxy implementation for the *logging* stage is very similar to that for *default* stage, except that it records the run-time call stack information and exports this information upon detecting an attack, which is shown in Figure 11. Run-time call stack information is recorded by pushing the function identifier into a separated and protected stack when the dynsw_prologue() function is executed and popping the function identifier out during the execution of the dynsw_epilogue() function. Note that the stack does not record all the functions under execution. Instead, only transformed functions (i.e., potentially vulnerable functions) are recorded. Once an attack is detected, the segmentation fault signal handler exports the current content of the stack into a file named CANDIDATE_LIST, which will be read by the next stage.

```
void dynsw_prologue( int func_id){
        /* push the function identifier into a separated and protected stack */
        stack_push( func_id);
}
void dynsw_epilogue( int func_id){
        /* pop the function identifier from the separated and protected stack */
        stack_pop();
}
INLINE void segv_handler( int signum, siginfo_t *siginfo, void *ucontext){
        stack_export(CANDIDATE_LIST);
        next_state();
}
void dynsw_init(){ install_segv_handler( segv_handler); }
```

Figure 11. *Logging* Stage Implementation

The *watching* stage protects buffers allocated by functions whose identifiers are recorded in the file CANDIDATE_LIST. It inserts the function identifiers obtained from the file CANDIDATE_LIST into a hash table during the execution of the dynsw_init() function. Whenever the dynsw_proxy() function is invoked (with a function identifier as an argument) by a wrapper function, it looks up the hash table to find the function identifier. If the identifier is found, the wrapper invokes the protected version of the transformed function. Otherwise, the original version is invoked.

Each protected version of a transformed function allocates buffers from the heap and protects them with guard pages. Moreover, it associates the function identifier with the starting address of the guard pages during each buffer allocation. With these associations, we can identify the function containing the overflowed buffer when a SIGSEGV signal is caught. The identifier of the function is then logged to a file named VULNERABLE_LIST, and the monitor program is notified to transit to the *protecting* stage so as to reduce further the number of buffers that need to be protected with guard pages.

However, if the program has more than one vulnerable function, our mechanism identifies them one by one by repeating the process of stage transition. In this case, the *watching* stage needs to protect not only the buffers recorded in the file CANDIDATE_LIST but also the ones residing in the previously-identified vulnerable functions.

Even though this stage is mainly responsible for identifying unidentified vulnerable functions; those that have been previously identified still have to be protected during this stage. For this reason, identifiers of function previously identified and recorded in the file VULNERABLE_LIST, as well as those contained in the file CANDIDATE_LIST, are inserted to the hash table g_hash, which is used in the dynsw_proxy() as described earlier. Figure 12 shows the implementation of this *watching* stage. Upon receiving an attack, the segmentation fault handler obtained the identifier of the vulnerable function through the association of function identifiers and guard pages. Then it searches for this function identifier in the hash table g_hash_v, which only contains identifiers of function previously identified as vulnerable. If the identifier is not found in the g_hash_v, which another vulnerable function is newly identified, the program inserts the identifier into the g_hash_v, exports it to the VULNERABLE_LIST and transits to the *protecting* stage. Otherwise, which means that this attack targeted on a previously identified vulnerable function, the program recovers from this attack and continue the execution as the *protecting* stage does. We describe how to do this later in the context of describing the *protecting* stage.

```
int dynsw_proxy( int func_id){
        return hash_find( &g_hash, func_id) ? G_VERSION : O_VERSION;
}
INLINE void segv_handler( int signum , siginfo_t *siginfo , void *ucontext){
        int func_id = mm_lookup( get_fault_addr( ucontext));
        if( hash_find( &g_hash_v, func_id)){
                fixup( ucontext);
        }
        else{
                hash_add( &g_hash_v, VULNERABLE_LIST);
                hash_export( &g_hash_v, VULNERABLE_LIST);
                next_state();
        }
}
void dynsw_init(){
        install_segv_handler( segv_handler);
        hash_init( &g_hash_v);
        hash_import( &g_hash_v, VULNERABLE_LIST);
        hash_init( &g_hash);
        hash_import( &g_hash, VULNERABLE_LIST);
        hash_import( &g_hash, CANDIDATE_LIST);
}
```

Figure 12. *Watching* Stage Implementation

The *protecting* stage protects fewer buffers than the *watching* stage. Specifically, it only protects the buffers allocated in the vulnerable functions, which were identified in the watching stage. The same as the *watching* stage, the *protecting* stage can execute through further buffer overflows by taking advantage of the idea of failure-oblivious computing. Since the guard pages prevent the memory data nearby the overflowed buffer from being overwritten, the program can execute through the buffer overflow without losing its correctness.

Figure 13 shows the proxy implementation of this stage. During the execution of dynsw_init(), it imports the identifiers of functions that have been identified as vulnerable to a hash table. This hash table is then used for determining the control flow whenever dynsw_proxy() is invoked. Upon detecting an attack, the segmentation fault handler figures out whether the target of the attack is a previously identified vulnerable function by checking if the faulting address belongs to one of the guard pages. If the faulting address does not belong to one of the guard pages, the program transits to the *logging and protecting* stage to repeat the process of identifying the vulnerable function; we will discuss this case later. Otherwise, the handler does not terminate the program, but instead ignores this segmentation fault and continues the execution.

However, simply returning from the signal handler will result in an endless loop since the instruction that causes the segmentation fault will be executed again. To solve this problem, the signal handler modifies the program counter by adding the length of the instruction that causes the segmentation fault. The program counter can be obtained from the argument of the signal handler. When a signal is caught, the kernel saves the program context, which consists of the register values, on the stack, and passes a pointer to the program context as an argument to the signal handler. When the signal handler returns, the modified program counter is also restored to the register and the program resumes the execution as if the segmentation fault never happened.

```
int dynsw_proxy( int func_id){
        return hash_find( &g_hash, func_id) ? G_VERSION : O_VERSION;
}
INLINE void segv_handler( int signum, siginfo_t *siginfo, void *ucontext){
        if( mm_lookup(get_fault_addr( ucontext)))
                fixup( ucontext);
        else
                next_state();
}
void dynsw_init(){
        install_segv_handler( segv_handler);
        hash_init( &g_hash);
        hash_import( &g_hash, VULNERABLE_LIST);
}
```

Figure 13. *Protecting* Stage Implementation

The *logging and protecting* stage can be regarded as a combination of the *logging* and the *protecting* stages. It collects rum-time call stack information while protects previously identified vulnerable functions at the same time. The implementation of this stage, which is shown in Figure 14, should be very easy to understand without further descriptions since both the *logging* stage and the *protecting* stage have been described earlier.

```
int dynsw_proxy( int func_id){
    return hash_find( &g_hash, func_id) ? G_VERSION : O_VERSION;
}
void dynsw_prologue( int func_id){
    stack_push( func_id);
}
void dynsw_epilogue( int func_id){
    stack_pop();
}
INLINE void segv_handler( int signum, siginfo_t *siginfo, void *ucontext){
    if( mm_lookup(get_fault_addr( ucontext)))
        fixup( ucontext);
    else
        next_state();
}
void dynsw_init(){
    install_segv_handler( segv_handler);
    hash_init( &g_hash);
    hash_import( &g_hash, VULNERABLE_LIST);
}
```

Figure 14. *Logging and Protecting* Stage Implementation

4 Evaluation

We have conducted a series of experiments to test the capability and evaluate the performance of our system. Table 2 lists five open source network servers we have included as test programs in our experiments. The vulnerability column shows CVE identifiers, which are unique and common identifiers for public known security vulnerabilities.

Table 2. List of Vulnerable Programs

Vulnerable Server	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

We transform the programs with TXL [31], which is a special-purpose programming language designed for creating, manipulating and rapidly prototyping language descriptions, tools and applications. Especially, we modify and extend Gemini [8], a tool which transforms source code to reposition stack-based buffers on the heap with TXL.

Table 3 lists the experimental environment. We recompile all the test programs listed in Table 2 and run them on a server machine with a 2.0 GHz Pentium 4, and 512MB of RAM, running Linux kernel 2.6.21. All performance evaluations are performed with benchmarks on a client machine with a 3.4 GHz Pentium 4, and 768MB of RAM, running Linux kernel 2.6.21. The two machines communicate via a 100MB Ethernet private network.

Table 3. Experiment Environment

Server (runs test servers)	Client (performs benchmarks)
Linux Kernel 2.6.21 Pentium 4 CPU 2.0GHz 512MB RAM Ethernet 100 MB	Linux Kernel 2.6.21 Pentium 4 CPU 3.4 GHz 768 MB RAM Ethernet 100 MB

4.1 Performance Evaluation

For each of the test programs, we evaluate the performance of both the original version and the transformed version. For the transformed version, the experiments are performed as follows. The monitor program launches the test programs at the *default* stage and waits for the requests. Then the benchmark is started on the client system for making requests and measuring performance of the server program. Upon getting the benchmark results, we send attack messages to crash the program on the server. As soon as the program detects the attack, it notifies the monitor program to transit to the next stage. Again, we start another benchmark for that stage. The same process is repeated until the program reaches the *protecting* stage.

4.1.1 Qpopper

Qpopper is a very 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 in a MDEF command with a long macro name.

We use Postal [33], a benchmark for measuring perfomance of SMTP and POP3 servers, to create POP3 seesions in a saturated manner to the server which has 2000 mailboxes with a total size of 50M bytes of messages. A typical POP3 session include logging on the server, listing mail messages, retrieving messages and/or deleting messages. The normalized results, which are shown in Figure 15, indicate that the performance degradation of the transformed Qpopper is quite low at a range from 1% to 3%.

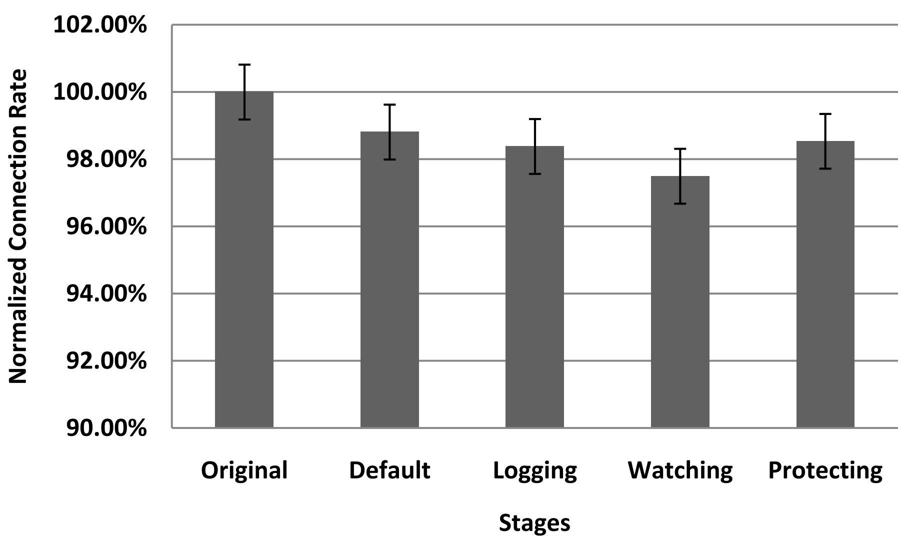

Figure 15. Performance Evaluation of Qpopper

4.1.2 Dproxy-nexgen

Dproxy-nexgen is a small caching domain name server. The vulnerability is a buffer overflow within the *dns_decode_reverse_name()* function of the dproxy-nexgen program, which allows remote attackers to execute arbitrary code by sending a crafted packet to UDP port 53.

The evaluation is conducted by measuring the average response time of looking up a cached domain name for 1000 iterations. The normalized result, which is shown in Figure 16, indicates that all the stages of transformed version have very low (below 2%) performance overhead.

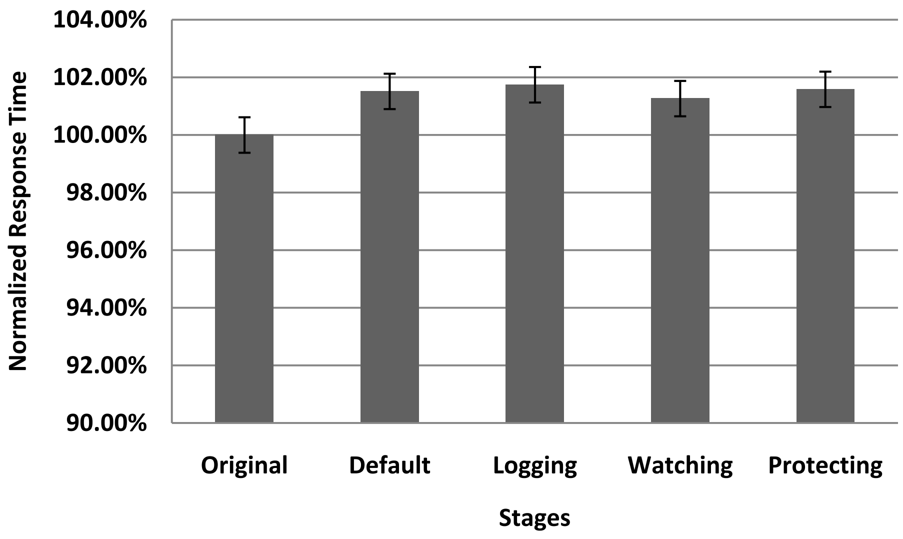

Figure 16. Performance Evaluation of dproxy-nexgen

4.1.3 ProFTPD

ProFTPD is a high-performance and highly configurable FTP server. The vulnerability is a buffer overflow within the *pr_ctrls_recv_request()* function in the src/ctrls.c file. This function calls *read()* to read *reqarglen* bytes from user input to a fixed length buffer. However, the *reqarglen* is also read from the user input and hence allows malicious users to control the program to read more data than the buffer can hold.

We evaluate the performance by fetching files of different sizes for 100 iterations. The normalized throughput, which is shown in Figure 17, indicates that the performance drops below 90% when fetching small files. However, as the file size grows, the overhead can be amortized and the performance degradation will become less perceptible.

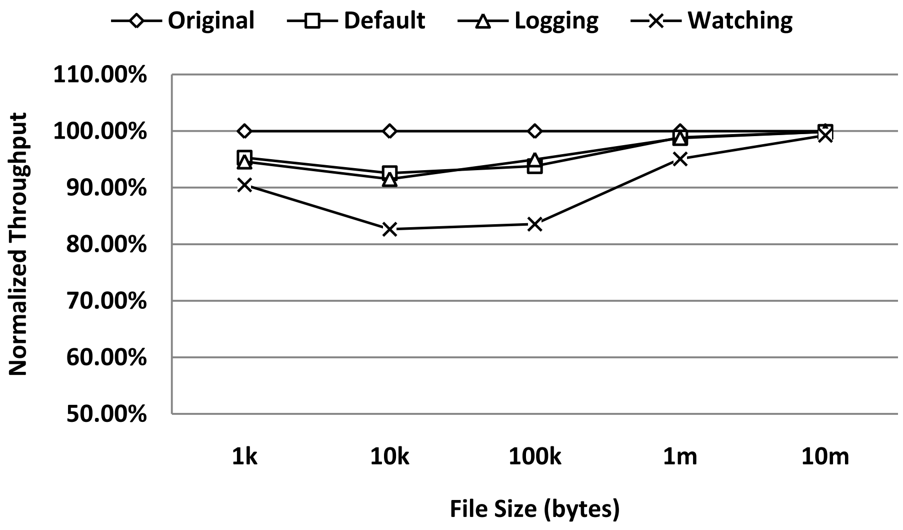

Figure 17. Performance Evaluation of ProFTPD

Notice that, the transformed ProFTPD does not transit to the *protecting* stage from the *watching* stage in this experiment. This is because the overwritten is happened in the *read*() system call. The Linux kernel handles this exception, which is caused by the guard page, by doing an early return to the user space instead of issuing a SIGSEGV signal to the program. As a result, the stage transition is not triggered. In this case, the program memory is not compromised and the program continues the execution without losing correctness.

4.1.4 Ghttpd

Ghttpd is 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 [32], a standard benchmark for web server, to evaluate performance of transformed ghttpd. Figure 18 shows the normalized results in terms of throughput and response time. As shown in the figure, the performance overhead is less than 3% in the *default, logging* and *protecting* stages. Even watching stage causes less than 5% overhead.

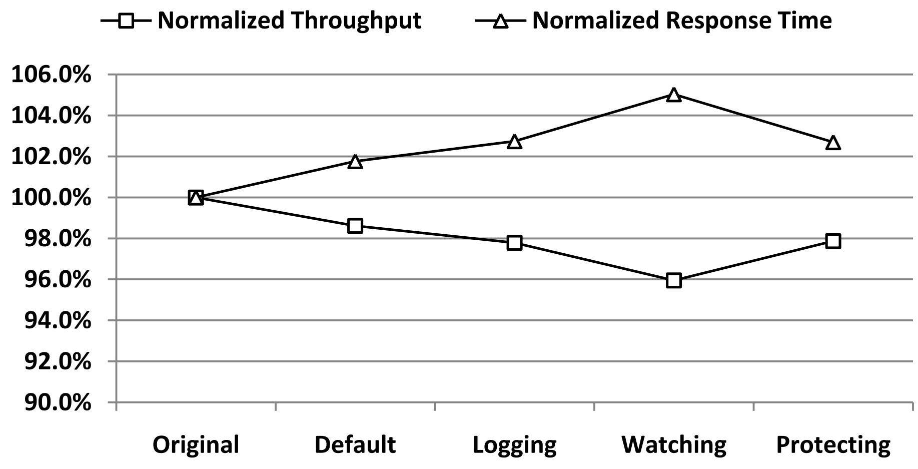

Figure 18. Performance Evaluation of ghttpd

4.1.5 Apache Tomcat Connector (mod_jk)

Apache Tomcat Connector (mod_jk) is 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.

Figure 19 shows the results of performance evaluation conducted by WebStone. As the figure indicates, all the stages of the transformed Apache program have a similar performance with the original version.

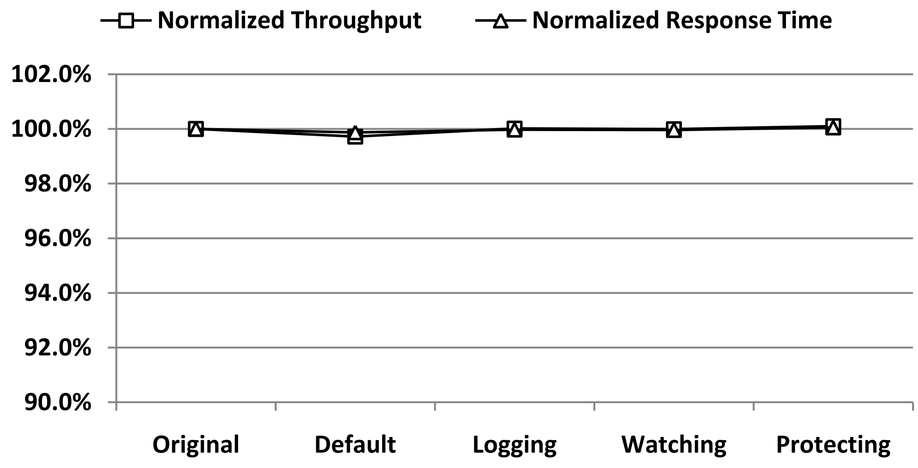

Figure 19. Performance Evaluation of Apache

4.2 Service Availability

To increase the responsiveness, Apache uses a single control process to maintain several spare worker processes, which are ready to serve incoming requests, so that clients do not have to wait for a new worker process to be forked before their requests can be served. For a vulnerable Apache server, serving an attack message as a request causes one of the worker process to crash. The server can not render useful service when the attack rate (i.e., the rate at which the attack messages are repetitively sent) is higher than a certain value.

To verify that the proposed mechanism has the ability to improve the service availability under repetitive attacks, we wrote a script to send attack messages at different rates and measure the performance degradation for both the original version and the transformed version (in the *protecting* stage) of Apache.

During the time we were performing this experiment, we noticed that the performance degradation varies with different numbers of client processes configured in the WebStone benchmark. Since WebStone client processes send requests to the web server in a saturated manner, we consider that the performance degradation variation is due to the competition between the attacker processes and the WebStone client processes. Therefore we modify the script to have the attack messages be uniformly distributed to 100 attacker processes to increase the parallelism of attacks.

Figure 20 shows the improvement of service availability, which is measured as the ratio of throughput under different attack rates when WebStone is configured to have 30 and 90 client processes. The result indicates that the transformed version still preserve from 60% to 70% of service availability while the original version can only render less than 10% of the service.

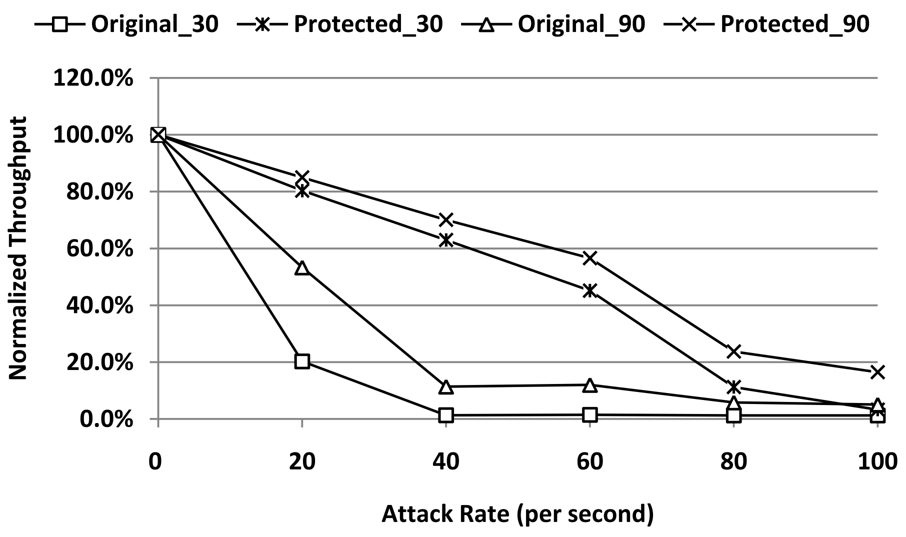

Figure 20. Availability Degradation of Apache under Repetitive Attacks

5 Related Work

Over the years, many techniques were proposed to address the buffer overflow vulnerability problem. Originally, most of the research efforts were put on the detection, either statically or dynamically, of the attacks. In recent years, more research efforts focused on recovering the attacked program in order to resist automated and repetitive attacks. In this section, we will describe those efforts.

5.1 Static Analysis Techniques

Static analysis techniques [9] [13] detects memory error problems at source code level. Unfortunately, analysis on some context sensitive code which requires program runtime information can not be done statically.

As a result, some techniques tend to generate too many false positives while other can miss real errors. Cyclone [11] and CCured [17] 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. However, manual source code modification is required for CCured to work with complex programs and for Cyclone to work with legacy C programs.

5.2 Dynamic Detection Techniques

Several [1] [22] dynamic techniques detect buffer overflow attacks by checking the integrity of control data (i.e., return address, frame pointers, etc). StackGuard [5] [6] places a canary value between local buffers and control data on a stack to check if the control data was corrupted due to buffer overflow. The canary value must be clobbered before corruption of the control data. Therefore, we can detect attacks by checking the integrity of the canary value upon function return. ProPolice [10] also uses canary values. In addition, it rearranges the stack layout to put arrays on the highest part of the stack frame so that other variables (if any) will not be corrupted when the buffers are overflowed. StackSheild [29] and RAD [3] 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 [4] augmented the GCC compiler to emit code that encrypts and decrypts pointers before and after they are stored in memory, respectively.

Therefore, an in-memory pointer overwritten by an attack can not be successfully decrypted without the decryption key. These techniques can be integrated with static analysis approaches to reduce the performance overhead. For example, there is no need to emit instrumentation code for functions without local buffers since no stack-based buffer overflows can happen in those functions.

Non-executable buffers [7] [19] [28] prevent execution of code on stacks or heaps with hardware assistance. An attacker may still inject the code into the buffers, but any attempt to execute that code will cause an exception. Address Space Layout Randomization (ASLR) [19] 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. Attackers may still overwrite the control data and redirect the program control flow, however, they have to guess the target address, and a wrong guess usually leads to a crash.

5.3 Automatic Recovery from Attacks

In recent years, automated attacks, which try to exploit the buffer overflow vulnerability in more fast and repetitive ways, have motivated the research on more effective protection mechanisms. The techniques described in the previous section can efficiently detect such attacks; however, they can not protect the victim processes from being compromised. Once an attack has been detected, they simply terminate the victim process to prevent further error propagation, and restart another instance if necessary. Fast and repetitive attacks cause a victim service to keep restarting and thus degrade the service availability.

Input-filtering is a general approach that can prevent the attacks before the process is compromised. DIRA [27] augmented the GCC compiler to log memory updates and track data dependency during the program execution. It logs memory update of global or static variables for recovering back the program after attack. Besides, it also intercepts some library functions to track propagation of external data for their attack identification algorithm. When an attack is detected, the program identifies the external input data that corresponds to the attack according to the data dependency. It then passes the data as attack signature to the front-end filter and rolls back the program to an earlier state where the input is not yet received. However, logging memory update and tracking data dependency degrade the performance of the program. Moreover, checkpointing only part of the program state can not always bring the program back to the correct state, and changes made to the file system can not roll back either.

TaintCheck [18] run programs in an emulation environment, which tags data derived from un-trusted 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 for the filter. The use of the emulation environment has an impact on the program performance and limits the practical usage.

Sidiroglou and Keromytis [24] [25] [26] 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. [20] [21] 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 [22] to augment the generated code to perform bounds checks and to store away or discard out of bounds writes [20] [21]. As a result, no memory data can be clobbered by buffer overflow. However, bounds checks downgrade the performance and make the approach impractical for many applications.

We propose a lightweight buffer overflow protection mechanism, which adopts the idea of failure-oblivious computing [20] [21]. However, unlike their work, our multi-stage design allows the program to achieve the same goal with little performance overhead.

6 Conclusions

Most techniques proposed to address the buffer overflow problem terminated the compromised process to cease error propagation. With repetitive attacks, however, such reaction is not desired for most of network services since keep restarting the process degrades the service availability to their intended users. This paper proposed a lightweight buffer overflow protection mechanism which recovers program from attacks. With guard page mechanism, the buffers are protected so that attackers can not compromise the victim process by overflowing the buffers. Therefore, the program can continue the execution without losing correctness and hence preserve the service availability. The proposed mechanism has very low impact on the performance of protected programs because it can only protect the buffers allocated in the vulnerable function.

References

[1] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-time Defense. Against Stack Smashing Attacks. In Proceedings of the 2000 USENIX Annual Technical Conference, pages 251-262, Jun. 2000.

[2] B. Perens. efence(3), April 1993 http://perens.com/FreeSoftware/ElectricFence

[3] T.-C. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In Proceedings of the 21st International Conference on Distributed Computing Systems, Apr. 2001.

[4] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. In Proceedings of the 12th USENIX Security Symposium, pages 91-104, Aug. 2003.

[5] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In Proceedings of the 7th USENIX Security Conference, pages 63-78, Jan. 1998.

[6] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In Proceedings of DARPA Information Survivability Conference and Exposition, pages 119-129, Jan. 2000.

[7] C. Dik. Non-Executable Stack for Solaris. Posted to comp.security.unix January 2, 1997.

[8] Christopher Dahn, Spiros Mancoridis. Using Program Transformation to Secure C Programs Against Buffer Over The 10th Working Conference on Reverse Engineering, British Columbia, Canada, November, 2003, pp.323-332.

[9] N. Dor, M. Rodeh, and M. Sagiv. Cssv: Towards a Realistic Tool for Statically Detecting all Buffer Overflows in C. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pages 155-167, Jun. 2003.

[10] H. Etoh and K. Yoda. Protecting from Stack-Smashing Attacks. Available at http://www.trl.ibm.com/projects/security/ssp, Jun. 2000.

[11] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: a Safe Dialect of C. In Proceedings of the USENIX Annual Technical Conference, pages 275-288, Jun. 2002.

[12] M. Kaempf. Smashing The Heap For Fun And Profit. Available at

http://doc.bughunter.net/buffer-overflow/heap-corruption.html

[13] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In Proceedings of the 10th. USENIX Security Symposium, pages 177-190, Aug. 2001.

[14] Z. Liang and R. Sekar. Automated, Sub-Second Attack Signature Generation: A Basis for Building Self-Protecting Servers. In Proceedings of the 12th ACM Conference on Computer and Communications Security, Nov. 2005.

[15] Z. Liang and R. Sekar. Automatic Generation of Buffer Overflow Attack Signatures:An Approach Based on Program Behavior Models. In Proceedings of the 21st Annual Computer Security Applications Conference, pages 215-224, Dec. 2005.

[16] Z. Liang, R. Sekar, and D. C. DuVarney. Automatic Synthesis of Filters to Discard Buffer Overflow Attacks: A Step Towards Realizing Self-Healing Systems. In Proceedings of USENIX Annual Technical Conference, pages 375-378, Apr. 2005.

[17] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In Proceedings of 29th. ACM Symposium on Principles of Programming Languages, pages 128-139, Jan. 2002.

[18] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In Proceedings of 12th Annual Network and Distributed System Security Symposium, Feb. 2005.

[19] The PAX team. Pax Address Space Layout Randomization. Available at http://pax.grsecurity.net.

[20] M. Rinard, C. Cadar, D. Roy, and D. Dumitran. A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors). In Proceedings of the 20th Annual Computer Security Applications Conference, pages 82-90, Dec. 2004.

[21] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In Proceedings 6th Symposium on Operating Systems Design and Implementation (OSDI), December 2004.

[22] O. Ruwase, M. Lam. A Practical Dynamic Buffer Overflow Detector. In Proceedings of the Network and Distributed System Buffer overflow Symposium, pages 159-169, Feb. 2004.

[23] H. Shacham, Matthew Page, B. Pfa, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In Proceedings of the 11th ACM Conference on Computer and Communications Security, pages 298-307, Oct. 2004.

[24] S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In Proceedings of the 12th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, pages 220-225, Jun. 2003.

[25] S. Sidiroglou and A. D. Keromytis. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. 8th Information Security Conference. September 2005.

[26] S. Sidiroglou, M. E. Locasto, S. W. Boyd and A. D. Keromytis. Building a Reactive Immune System for Software Services. In USENIX Annual Technical Conference, Apr. 2005.

[27] A. Smirnov and T. C. Chiueh. DIRA: Automatic Detection, Identification and Repair of Control-Hijacking Attacks. In Proceedings of 12th Annual Network and Distributed System Security Symposium, Feb. 2005.

[28] Solar Designer. Non-Executable User Stack. http://www.openwall.com/linux/.

[29] StackShield. http://www.angelfire.com/sk/stackshield, Jan. 2000.

[30] J. Xu, P. Ning, C. Kil, Y. Zhai and C. Bookholt. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In Proceedings of 12th ACM Conference on Computer and Communications Security, pages 223-234, Nov. 2005.

[31] J. R. Cordy, T. R. Dean, A. J. Malton and K. A. Schneider, "Source Transformation in Software Engineering using the TXL Transformation System", Journal of Information and Software Technology44(13), 827--837 (2002).

[32] WebStone. Available at http://www.mindcraft.com/benchmarks/webstone

[33] Postal. Available at http://www.coker.com.au/postal