July 13, 2000
RT0371
Security 9 pages

Research Report

ProPolice: Protecting from stack-smashing attack

Hiroaki Etoh and Kunikazu Yoda

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

ProPolice: Protecting from stack-smashing attacks

Hiroaki Etoh and Kunikazu Yoda
IBM Research Division, Tokyo Research Laboratory,
1623-14 Shimotsuruma, Yamato, Kanagawa 242-8502, Japan
{etoh,yoda}@jp.ibm.com

June 19, 2000

Abstract

A stack-smashing attack is an attack method that causes services to be stopped and also allows an attacker to intrude into a system. It uses a well-known vulnerability of applications called the buffer overflow vulnerability. The stack-smashing attack is the most common attack method reported in UNIX security news reports. In 1999, the vulnerability of IIS 4.0, the popular web server for the Microsoft Windows operating system, was reported. There were more than 1.5 million systems attacked throughout the world.

This paper presents a systematic solution to the problem of buffer overflow attacks. Our approach called *ProPolice* provides a protection method that automatically inserts protection code into an application at compilation time. The main characteristics of *ProPolice* are low performance overhead of the protection code, protecting against different varieties of stack-smashing attacks, and supporting various processors.

1 Introduction

A stack-smashing attack is an attack method that causes services to be stopped and also allows an attacker to intrude into a system. It is based on the vulnerability of an application called the buffer overflow vulnerability. It is the most common vulnerabilities reported from UNIX security news announcements. In 1999, a vulnerability in IIS 4.0, the popular web server for the Microsoft Windows operating system, was exploited by eEye[4]. There were more than 1.5 million systems attacked throughout the world.

Most applications written in C use a buffer, which is a memory block that holds several instances of the same data type, normally character arrays, on the stack to temporarily hold the intermediate results of string operations. A stack-smashing attack overflows such a buffer by providing a longer string than the actual size of the buffer. This causes the destruction of the contents beyond the buffer, where such contents may include the return address of the caller function and function pointers.

This paper presents a systematic solution to the problem of buffer overflow attacks. Our approach called *ProPolice* is based on a protection method that automatically inserts protection code into an application at compilation time. The main characteristics are low performance overhead of the protection code, protecting against different varieties of stack-smashing attacks, and supporting various processors. Especially important is that there is little overhead during the execution of numerical operations. These are CPU intensive tasks, so people want to avoid any overhead for the sake of protection.

Section 2 classifies attack methods to provide background for explaining how our method protects against the attack. Section 3 describes related work for defense against stack-smashing attacks. Section 4 provides the protection method and explains how it optimizes the performance overhead. We'll show some experimental results in Section 5. Finally, Section 6 presents conclusions and discusses issues for future work.

2 Attack Scenarios and their Classification

The buffer overflow vulnerability appears where an application needs to read external information such as a character string, the receiving buffer is relatively small compared to the possible size of the input string, and the application doesn't check the size. The buffer allocated at run-time is placed on a stack, which keeps the information for executing functions; such as local variables, argument variables, and the return address. The overflowing string can alter such information. This also means that an attacker can change the information as he wants to. For example, he can inject a series of machine language commands as a string that also leads to the execution of the attack code by changing the return address to the address of the attack code. The ultimate goal is usually to get control of a privileged shell by such methods.

Figure 1 shows a typical stack structure after a function is called. The stack pointer points at the top of stack, which is at the bottom in the figure. The programming language C uses the area from the top of the stack in the following order: local variables, the previous frame pointer, the return address, and arguments of the function. This data is called the frame of the function, which represents the status of the function. The frame pointer locates the current frame and the previous frame pointer stores the frame pointer of the caller function.

The function foo (see Figure 2) is a vulnerable function, which produces the stack structure such as shown in Figure 1. It reads the content of the environment variable "HOME" into the "buffer" which has a size of 128 bytes. Since the function strcpy doesn't check the size of the output, it can copy more than 128 bytes of data to the "buffer". Imagine the "HOME" variable has this string: 128 bytes of 41, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, and 3. This will assign 128 character 'A's, 0x01010101, 0x02020202, and 0x03030303 into the "buffer", "lvar", the previous frame pointer, and the return address respectively. (We assume that 32-bit variables are used by default and that C language notation is used.) When the function foo finishes it's operations and returns to the caller based on this memory arrangement, it will go back to the address 0x03030303, which isn't the caller address. If malicious code is located at the address, it is executed with the same privilege level as the application.

```
void foo()
{
   long *lvar;
   char buffer[128];
   .......
   strcpy (buffer, getenv ("HOME"));
   .......
}
```

Figure 2: Sample function having Buffer Overflow Vulnerability

We will now introduce a classification of attack methods, how an attacker acquires control of the application. In the first category the target of the attack is to show in the stack. The following lists the data stored in this area and describes the attack method used.

- return address

  It is the most popular point of attack by changing the value of the return address to the address of malicious code.

- local variables

- argument variables

  The function pointer variable is a another target for acquiring the control. Assigning the function pointer variable of an argument or a local variable to the attack code is a typical attack method. In this case, the vulnerable place can be found by checking the source program.

  There is a third attack method using redirection. Assume that a variable is declared as the pointer of the structure that holds a function pointer. This is vulnerable to attack, but it is hard to find it without traversing the pointer. However , an attacker can create a fake structure that has the same contents as the original except for the function pointer.

  There is a possibility that changing a pointer variable which is not a function pointer variable will acquire the control of the application. In the case of the function foo from Figure 2, the pointer variable "lvar" can be changed to point to the location of the return address. If there is a

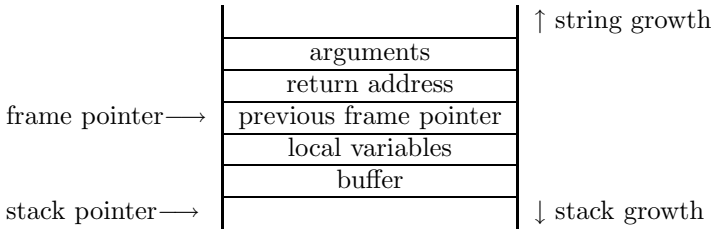

Figure 1: Stack Structure

statement that modifies the value pointed to by "lvar" after the "strcpy" statement, changing the return address may be possible.

- previous frame pointer

  Attacking the previous frame pointer can also take the control of the application. The connection between the previous frame pointer and the return address is based on the following dependencies:

  – the location of the return address is determined by the frame pointer.

  – the frame pointer is assigned to the value of the previous frame pointer at the time of function return.

  An attacker can create a fake frame that has the return address pointed to attack code. He can also change the previous frame pointer to the address of the fake frame. When the function returns to the caller function, the frame pointer will point to the fake frame according to the second dependency. It means that the return address could be changed by the attacker.

3 Related works

Several projects have addressed the buffer overflow problem with different approaches. One approach[10] is to eliminate vulnerable code from a source program and to help the application be made safe from the problem. For example, there is an auditing tool that helps automate source code review for security[10]. It helps to eliminate the use of dangerous functions: such as strcpy, gets, etc., but the tool has the limited vulnerability detection in that it can't check the boundaries by a pointer variable.

Another approach provides protection methods against the potential vulnerability of program code. We have defined four categories according to how they protect against an attack.

1. Avoidance of leakage from an array

   Array bounds checking for C [7] and memory access checking [6] are protection methods that prevent access outside the region allocated for an array. Therefore, these methods are the most secure methods. However, the protection overhead is expensive compared to non-protected code; a slowdown of more than two times in comparison to optimized code is common.

2. Prohibition on the execution of attack code

   "Solar Designer" developed a Linux patch that makes the stack region non-executable [3], so that attack code stored on the stack cannot be executed. This approach has the advantages of no performance overhead and no source code is required. However, it has drawbacks in that it relies on the features of the operating system and the processor, specifically the capability of marking the stack region as non-executable. It doesn't protect every regions described in Section 2, so there is still a possibility that an attacker can take control, by putting the attack code somewhere besides into the stack, such as into a statically allocated buffer and changing the return address to point to the code.

   Janus [5] designed a secure environment for confinement of applications by restricting the programs' access to the operating system. It protects the privileged operations, such as executing `/bin/sh` in privileged mode, from attack code not just on the stack, but also in static regions. It also relies on the features of the operating system, which must provide debug information such as `strace`.

3. Protection not to pass the control of execution to code that has been attacked

   Snarskii has developed a FreeBSD patch [9] that implements a stack integrity check to detect buffer overflows. This is a non-portable implementation embedded in `libc`.

   StackGuard [2], StackShield [11], and libsafe [1] provide a portable, general protection method. Libsafe provides a solution, which is based on a middleware software layer that intercepts all function calls made to library functions that are known to be vulnerable; such as `gets`, `strcpy`, so on.

   StackGuard is the base system of our system, it detects and defeats stack smashing attacks by protecting the return address on the stack from being altered. The "XOR Random canary" method places the xor value of the return address and a random number next to the return address when a function is called and check if the value is preserved before the function returns.

   StackShield copies the return address in an unoverflowable location, the beginning of the static data, when the function begins execution and check if the two values are preserved before the function returns.

   We will compare each of these techniques and our method in Section 4.5.

4 Stack Protection Method

As described in the previous section, there are four areas that should be protected from a stack-smashing attack: the location of the arguments, the return address, the previous frame pointer, and the local variables.

A guard variable is introduced for preventing change in the first three areas. This technique was originally devised in the StackGuard project [2] and it is designed to insert the guard immediately after the return address. The difference in our method is the location of the guard and the protection of function pointers. The guard is inserted next to the previous frame pointer and it is prior to an array, which is the location where an attack can begin to destroy the stack.

We'll illustrates the method using a source code translation only for the purpose of a concise explanation. Given the source code of a function, a preprocessing step will insert the following fragments of code into the appropriate positions: declaration part of local variables, the entry points, and the exit points respectively.

- declaration part of local variables

```
volatile int guard;
```

- the entry point

```
guard = guard_value;
```

- the exit point

```
if (guard != guard_value) {
      /* output error log */
      /* halt execution */
}
```

The modified source program appears as Figure 3. Note that the guard must be located before a string buffer. The entry statement stores a value that is not known by the attacker and the exit statement verifies it. Unless the guard is preserved, it will cause an alert to the system and record information in the logging database.

```
void foo()
{
        volatile int guard;
        char buf[128];

        guard = guard_value;
        .......
        strcpy (buf, getenv ("HOME"));
        .......
        if (guard != guard_value) {
            /* output error log */
            /* halt execution */
        }
}
```

Figure 3: After the Protection Code has been Added

We introduced the source code translation to explain how the protection is done. In fact, it it difficult to implement, because a function usually has many exit points and the guard must be located before the string buffer on the stack. We have implemented our system using the intermediate code translator of the `gcc` compiler.

According to the specifications of the intermediate language, there is only one exit point per function and the location of each variables has been decided.

T main requirement of the guard value is that it must be a value that an attacker can't know. If he knows the value, the attacker could fill the part of the stack between the "buffer" and the previous frame pointer with the value. This would pass the verification test of the guard value, and then he could change the return address.

We chose a random number as the guard value. The random number is calculated at the initialization time of the application, which cannot be discovered by a non-privileged user. The number must be an unpredictable random number. Linux has a random number generator. It is implemented as a device, which name is /dev/urandom or /dev/random. It uses environmental noise to generate the number; thus, it is suitable to supply an unpredictable number.

4.1 Safety Function Model

We introduce a safety function model, which involves a limitation of stack usage 4 in the following manner:

- the location **(A)** has no array or pointer variable
- the location **(B)** has arrays or structures that contains an array
- the location **(C)** has no array

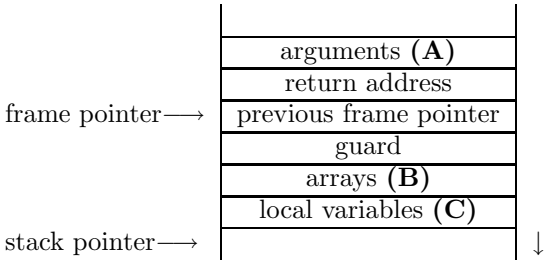

Figure 4: Safe Frame Structure

This model has the following properties:

1. The memory locations outside of a function frame cannot be damaged when the function returns.

   The location **(B)** is the only vulnerable location where an attack can begin to destroy the stack. Damage outside of the function frame can be detected by the verification of the guard value. If damage occurs outside of the frame, the program execution stops.

2. An attack on pointer variables outside of a function frame will not succeed.

   The attack could only succeed if the following conditions were satisfied: (1) the attacker changes the value of the function pointer, and (2) he calls a function using the function pointer. In order to achieve the second condition, the function pointer must be visible from the function, but our assumption says this information is beyond the function scope. Therefore, the second condition can't be satisfied, and the attack will always fail.

3. An attack on pointer variables in a function frame will not succeed.

   The location **(B)** is the only vulnerable location for a stack-smashing attack, and the damage goes away from area **(C)**. Therefore, the area **(C)** is safe from the attack.

4.2 Pointer protection

Figure 5 shows another vulnerable function, which an attacker may use to take control by replacing the function pointer with the address of attack code.

```
void bar( void (*func1)() )
{
        void (*func2)();
        char buf[128];
        .......
        strcpy (buf, getenv ("HOME"));
        (*func1)(); (*func2)();
}
```

Figure 5: Attack Against a Function Pointer

In order to protect function pointers from stack-smashing attacks, we change the stack location of each variables to be consistent with the safe function model.

The "C" language doesn't have restrictions on the order of local variables, but it doesn't allow changing the location of arguments. The restriction on arguments can be addressed by making a new local variable, copying the argument "func1" to it, and changing the reference

to "func1" to use the new local variable. Figure 6 shows the result of the translation.

```
void bar( void (*tmpfunc1)() )
{
        char buf[128];
        void (*func2)();
        void (*func1)(); func1 = tmpfunc1;
        .......
        strcpy (buf, getenv ("HOME"));
        (*func1)(); (*func2)();
}
```

Figure 6: Protection from Function Pointer Attack

4.3 Optimization

In this section, we'll discuss optimization techniques for reducing the overhead of the guard implementation. The following assumptions are introduced for that purpose. When these assumptions are met, some guard code can safely be removed.

Assumption 1 The source code has the proper type declarations and follows the type conversion rules.

For example, an integer variable always stores integer values, and does not store string values. Such erroneous operations often causes memory protection errors during the execution of a program.

Assumption 2 Only character arrays can cause the buffer overflow.

The buffer overflow happens during the assignment operation to an array without a boundary check. The operation commonly uses a terminator for the detection of the boundary. Most terminators are used only by string functions; examples are a null character or a newline character for the `gets` function.

If Assumption 2 is met in addition to Assumption 1, the guard protection can be omitted except for the functions that declare string buffers as local variables or as arguments. This is expected to reduce the overhead, especially for numerical processing, because those functions usually use numerical arrays and a few functions are called many times in a short period.

4.4 Limitation

The stack protection method is achieved by the program translation, which converts a vulnerable function to a non-vulnerable function. The conversion can't be always successful in some cases.

If a structure includes both a pointer variable and a character array, the pointer can't be protected, because changing the order of structure elements is prohibited.

There is another limitation on keeping pointer variables safe. It is when an argument is declared as a variable argument, which is used by a function with a varying number of arguments of varying types. The usage of pointer variables can't be determined at compilation time, but it can be determined only during execution.

4.5 Comparison of Protection Techniques

Figure 7 shows the comparison of protection techniques, which prevent to pass the control of execution to code that has been attacked.

The protection effectiveness describes the protection coverage of the protected region for the stack and the coverage against string functions causing buffer overflow. The entry marked "No" in the protected region means that the particular region cannot be protected and vulnerable programs could be found in the future. For example, the exploit program for the program called "SuperProbe 2.11" attacks the function pointer variable pointed at from the argument variable, so StackGuard cannot block the attack and the attacker can gain access to the root shell.

Libsafe doesn't protect all of the string functions, nor does it from protect from the string operations that are using pointers directly; for example, the exploit program for the program called "xterm distributed with RedHat5.2" uses a buffer overflow in the tgetent() function of libtermcap.

The implementation characteristics shows that only our system provides a protection mechanism for a variety of operating systems and processors.

Description	None	ProPolice	libsafe	StackGuard	StackShield
Protection Effectiveness					
return address	No	Yes	Yes	Yes	Yes[1]
prev. frame pointer	No	Yes	Yes	No	Yes[1]
argument	No	Yes	Yes	No	No
local variable	No	Yes[2]	No	No	No
string operation coverage	No	All	Not all	All	All
Implementation characteristics					
OS independence	–	Yes	No[3]	Maybe[4]	Maybe[4]
Processor independence	–	Yes	Yes	No	No
Other Characteristics					
performance overhead	None	Very low	Very low	Low	Low
source code needed	–	Yes	No	Yes	Yes

1 Protects only the function within a limited depth of function calls

2 It cannot protect a pointer variable in a structure that contains a vulnerable string

3 Needs dynamic link library

4 Intel processor only

Figure 7: Summary of Detection Technique Characteristics

5 Evaluation

The result of penetration tests are shown Figure8. It illustrates the application name which has a known buffer overflow vulnerability, its description, the result of exploits without any protection, and the result of exploits with our protection. Their first three are exploited by attacks on the return address and the last one is exploited by an attack on the arguments which point to a structure with a function pointer.

This is not a comprehensive exploit list, but enough to verify the protection method works well. It showed that all tests terminated with a message that a stack-smashing attack had been detected, they didn't invoke a root shell, and they didn't terminated abnormally.

5.1 Performance Overhead

The guard mechanism imposes an additional cost in program execution. The overhead was defined by Crispin[2] as follows: it is the ratio of the CPU time of a guarded function call per the base cost of the function call. The overhead of our system varies according to the presence of local character arrays. It is obvious that our optimized method has no overhead in the applications that have no local character arrays, such as integer sorting programs, linear programming systems, and so on.

The insertion overhead for a guard variable depends on the size of the function. Program 9 is introduced to measure the upper bound of the overhead. It is written to be an actual string program and to be as small as possible.

```
int test()
{
   char buf[128];
   strcpy(buf, "1234567890");
   return strncmp(buf, "1234", 4);
}
```

Figure 9: Program to Estimate the Upper Bound of the Overhead

The experiment was performed on a 600 MHz Pentium III with 512 K of level 2 cache, and 256 M of main memory. The time is based on 50,000,000 runs and is given in seconds.

original run time	our method run time	overhead (%)
4.67	5.05	8%

It shows an 8% overhead on function calls, which should be the upper bound on the real costs of running programs under this protection system. The overall overhead of guarded pro-

Exploit Program	Description	Result of Attack	Protected Result
xlockmore 3.10	Lock an X window display	root shell	terminated
Perl 5.003	Perl script language	root shell	terminated
elm 2.003	ELM mail user agent	root shell	terminated
SuperProbe 2.11	Probes video hardware	root shell	terminated

Figure 8: Penetration Resistance

grams varies with how many functions are called that have character array. Figure 10 shows a program's name, its description, the number of functions declared, and the number of functions used with character arrays. In most cases, the usage of a character array is less than 10% of the functions. It isn't the same as the ratio of the number of functions executed, but there is a some correlation between them.

The copy overhead of an argument is more expensive than the insertion overhead for a guard variable, if the argument is a structure that contains a character array or a pointer variable. Figure 11 shows the number of functions which arguments contain a character array, and the number of functions which arguments contain a pointer variable for each program. There are no structures being passed as parameters to function calls. Therefore, for each pointer argument, the overhead is at most one copy operation from an argument to a local variable.

Program	with string	with pointer
perl 5.005	0	0
ctags 3.4	0	0
imap 4.7-5	0	0
XFree86 3.3.6	0	0
kernel 2.2.14	0	0
egcs 1.1.2	0	0
glibc 2.1.3	0	0

Figure 11: The Number of Copying Overhead Operations for Arguments

Figure 12 shows the run-time cost of three real-world applications to compare the overhead of our method, libsafe, and StackGuard. The applications are `perlbench` (a CPU-bound program) mesuring the time of several operations [8], `ctags` (an I/O-bound program) indexing egcs-1.1.2 directory, and `imapd` transmitting 100 email messages of size two kilobyte each. The programs are selected from programs that mainly process string operations to illustrate the upper bound of the overall overhead. The execution times are based on 100 runs with associated 95% confidence intervals. The times are elapsed times using `/bin/time`. By looking at Figure 13 showing the comparison of each system's performance overhead, it can be seen that our method is the most efficient.

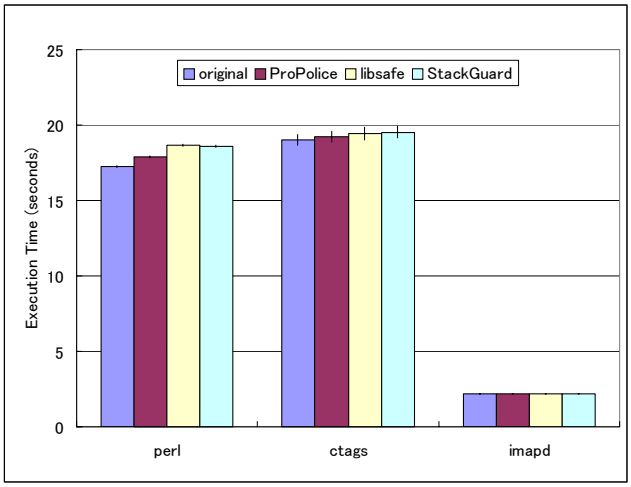

Figure 12: Mean Execution Times

	perl	ctags	imapd
ProPolice	4%	1%	0%
libsafe	8%	2%	0%
StackGuard	8%	3%	0%

Figure 13: Comparison of performance overhead

6 Conclusions

We have described the stack area where the control of an application can be captured by a stack-smashing attack. We have described our protection method, which is based on the canary method of StackGuard[2] to protect the location of the return address, extending the protection to the location of the previous frame pointer, the arguments, and the local variables.

Our method achieves good performance on several application benchmarks. We have de-

Program	Description	function count	w/ string	ratio(%)
perl 5.005	Perl script language	9590	54	0.6%
ctags 3.4	Tag files generator	850	4	0.5%
imapd 4.7-5	ELM mail user agent	915	259	6.4%
XFree86 3.3.6	X window system	22198	1145	5.1%
kernel 2.2.14	Linux kernel	3881	112	2.8%
egcs 1.1.2	GNU compiler system	27019	244	0.9%
glibc 2.1.3	Library	719	106	14.7%

Figure 10: The Usage Counts for Character Buffers

scribed the reason, which is that the number of functions vulnerable to buffer overflow is relatively small compared to the total number of functions used.

We have implemented our system as a intermediate language translator for **gcc**, which means the implementation is independent of the operating systems and the processors used. We believe that the minimal performance overhead and its universal applicability makes it the best defense system for workstations, personal digital assistants(PDA), and cellular telephone system.

References

[1] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000. to be appeard.

[2] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, A. G. Steve Beattie, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings in the 7th USENIX Security Symposium*, January 1998.

[3] S. Designer". Non-executable user stack. http://www.false.com/security/linux/.

[4] "eEye-Digital Security Team". Iis4.0 remote exploit. http://www.eeye.com/, 1999.

[5] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *In Proceedings of the 6th USENIX Security Symposium*, 1996.

[6] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*. 1992.

[7] R. Jones and P. Kelly. Bounds Checking for C. http://www-ala.doc.ic.ac.uk/ phjk/BoundsChecking.html, July 1995.

[8] Perlbench. http://www.metacard.com/perlbench.html.

[9] A. Snarskii. FreeBSD stack integrity patch. ftp://ftp.lucky.net/pub/unix/local/libc-letter, 1997.

[10] "The Software Security Group. Its4: Open source software security tool. http://www.rstcorp.com/its4/.

[11] "Vendicator". Stack shield: A "stack smashing" technique protection tool for linux. http://www.angelfire.com/sk/stackshield/.