Performance Overhead of Buffer Overflow Prevention Tools
Kathy P. Huynh

ABSTRACT
For the past several years, buffer overflow attacks have been the main method of compromising a computing system’s security. Many of these attacks have been devastatingly effective, allowing the attacker to attain administrator privileges on the attacked system. This paper gives a brief description of what buffer overflow is and presents four buffer overflow prevention tools: Libsafe, Stack Guard, Stack Shield and Stack-smashing Protector (SSP). Then a comparison of performance overhead is done on these four tools.

1. INTRODUCTION
Buffer overflows constitute a major threat to the security of computer systems today. A buffer overflow exploit is both common and powerful, and is capable of rendering a computer system totally vulnerable to the attacker. As reported by CERT, 11 out of the 20 most widely exploited attacks have been found to be buffer overflow attacks [1]. More than 50% of CERT advisories [2] for the year 2001 reported buffer overflow vulnerabilities.

Vulnerabilities Reported to CERT

![Vulnerabilities Reported to CERT](image)

Figure 1: Software vulnerabilities reported to CERT 1995-2001

In November 1988, many organizations had to cut themselves off from the Internet because of the "Morris worm," which was a program written by 23-year-old Robert Tappan Morris to attack VAX and Sun machines. By some estimates, this program took down 10% of the entire Internet. In July 2001, another worm named "Code Red" eventually exploited over 300,000 computers worldwide running Microsoft's IIS Web Server. In January 2003, the "Slammer" worm (also known as "Sapphire") exploited vulnerability in Microsoft’s SQL Server 2000 software, disabling parts of the Internet in South Korea and Japan, disrupting Finnish phone service, and slowing many U.S. airline reservation systems, credit card networks, and automatic teller machines. All of these attacks -- and many others -- exploited a vulnerability called a buffer overflow.

Buffer overflow exploits come in various flavours. The simplest and also the most widely exploited form of attack changes the control flow of the program by overflowing some buffer on the stack so that the return address or the saved frame pointer is modified. This is commonly called the “stack smashing attack” [3]. Other more complex forms of attack may not change the return address but attempt to change the program control flow by corrupting some other code pointer (such as function pointers, GOT entries, longjmp buffers, etc.) by overflowing a buffer that may be local, global or dynamically allocated. Many common forms of buffer overflow attack are described in [4].

2. BUFFER OVERFLOW
A buffer is a contiguous allocated chunk of memory that holds more than one instance of the same data type, such as an array or a pointer in C. In C and C++, there are no automatic bounds checking on the buffer, which means a user can write past the end of a buffer. Since buffers are created to contain a finite amount of data, the extra information can overflow into adjacent buffers, corrupting or overwriting the valid data held in them. In buffer overflow attacks, the extra data may contain codes designed to trigger specific actions, in effect sending new instructions to the attacked computer that could, for example, damage the user's files, change data, or disclose confidential information. Let’s look a simple C/C++ code snippet that overruns a buffer.

```c
int main () {
    int buffer[10];
    buffer[20] = 10;
}
```

The above C program is a valid program, and every compiler can compile it without any errors. However, the program attempts to write beyond the allocated memory for the buffer, which might result in unexpected behavior.

3. BUFFER OVERFLOW ATTACK
A buffer overflow attack, the objective of the attacker is to use the vulnerability to corrupt information in a carefully designed way in order to execute attack code previously planted by the attacker. If this succeeds, the attacker has effectively hijacked control of the program. Once control is transferred to the attack code, it grants unauthorized access to the attacker. Typically the attack code just spawns a shell, which allows the attacker to execute arbitrary commands on the system.
A buffer overflow attack may be local or remote. In a local attack the attacker already has access to the system and may be interested in escalating his/her access privilege. A remote attack is delivered through a network port, and may achieve simultaneously both gaining unauthorized access and maximum access privilege. The most general form of security attack achieves two goals:
- Inject the attack code, which is typically a small sequence of instructions that spawns a shell, into a running process.
- Change the execution path of the running process to execute the attack code.

By far, the most popular form of buffer overflow exploitation is to attack buffers on the stack, referred to as the stack smashing attack [9].

4. SMASHING THE STACK

One classification of buffer overflow attacks depends on where the buffer is allocated. If the buffer is a local variable of a function, the buffer resides on the run-time stack. This is the type of attack examined in Levy's article [11], and it is by far the most prevalent form of buffer overflow attack.

When a function is called in a C program, before the execution jumps to the actual code of the called function, the activation record of the function must be pushed on the run-time stack. In a C program the activation record consists of the following fields:

1. space allocated for each parameter of the function;
2. the return address;
3. the dynamic link;
4. space allocated to each local variable of the function.

The function must be able to access its parameters and local variables. This requires that during the execution of the function a register hold the base address of the activation record of the function, i.e. the address of the dynamic link field. Parameters are below this address on the stack, and local variables above. When the function returns, this register must be restored to its previous value, to point to the activation record of the calling function. To be able to do this, when the function is called the value of this register is saved in the dynamic link field. Thus the dynamic link field of each activation record points to the dynamic link field of the previous activation record on the stack, which in turn points to the dynamic link field of the previous activation record, and so on, all the way to the bottom of the stack. The first activation record on the stack is that of main(). This chain of pointers is called the dynamic chain.

In many C compilers the buffer grows towards the bottom of the stack. Thus if the buffer overflows and the overflow is long enough the return address will be corrupted, (as well as everything else in-between, including the dynamic link.) If the return address is overwritten by the buffer overflow so as to point to the attack code, this will be executed when the function returns. Thus, in this type of attack, the return address on the stack is used to hijack the control of the program.

Overwriting the return address, as explained above, gives the attacker the means of hijacking the control of the program, but where should the attack code be stored? Most commonly it is stored in the buffer itself. Thus the payload string which is copied into the buffer will contain both the binary machine language attack code as well as the address of this code which will overwrite the return address.

There are a few difficulties that the attacker must overcome to carry out this plan. If the attacker has the source code of the attacked program it may be possible to determine exactly how big the buffer is and how far it is from the return address, determining how big the payload string must be. Also, the payload string cannot contain the null character since this would abort the copying of the payload into the buffer. Some copying routines of the C library use carriage returns and new lines as a delimiter instead, so these characters should also be similarly avoided in the payload string.

5. BUFFER OVERFLOW PREVENTION TOOLS

As we mentioned earlier, the most common type of buffer overflow is stack smashing. In this section, we present four publicly available buffer overflow prevention tools. These tools commonly deal with stack smashing attacks.

5.1 Libsafe

Libsafe is an implementation of vulnerable copy functions in C library such as strcpy(). In addition to the original functionality of those functions, it imposes a limit on the involved copy operations such that they do not overwrite the return address. The limit is determined based on the notion that the buffer cannot extend beyond its stack frame. Thus the maximum size of a buffer is the distance between the address of the buffer and the corresponding frame pointer. Libsafe is implemented as a shared library that is preloaded to intercept C library function calls. A substitute version of the corresponding function implements the original function in a way that ensures that any buffer overflows are contained within the current stack frame, which prevents attackers from overwriting the return address and hijacking the control flow of a running program.

The true benefit of using Libsafe is protection against future attacks on programs not yet known to be vulnerable. Libsafe does not require changes to the OS, it works with existing binary programs, and it does not need access to the source code of defective programs, or recompilation or off-line processing of binaries [9].

Libsafe is available for download at http://www.research.avayalabs.com/project/libsafe.

5.2 Stack Shield

Stack Shield is a GNU C compiler extension that protects the return address [8]. In the current version 0.7 it implements three types of protection, two against overwriting of the return address and one against overwriting of function pointers.

The Global Ret Stack protection of the return address is the default choice for Stack Shield. It is a separate stack for storing the return addresses of functions called during execution. Whenever a function call is made, the return address being pushed onto the normal stack is at the same time copied into the Global Ret Stack array. When the function returns, the copy on
the Global Ret Stack replaces the return address on the normal stack.

The Ret Range Check is somewhat simpler but faster version of Stack Shield’s protection of return addresses. It uses a global variable to store the return address of the current function. Before returning, the return address on the stack is compared with the stored copy in the global variable. If there is a difference the execution is halted.

Stack Shield also aims to protect function pointers from being overwritten. The idea is that function pointers normally should point into the text segment of the process’ memory. That’s where the programmer is likely to have implemented the functions to point. If the process can ensure that no function pointer is allowed to point into other parts of memory than the text segment, it will be impossible for an attacker to make it point at code injected into the process, since injection of data only can be done into the data segment, the BSS segment, the heap, or the stack. Stack Shield adds checking code before all function calls that make use of function pointers. A global variable is then declared in the data segment and its address is used as a boundary value. The checking function ensures that any function pointer about to be dereferenced points to memory below the address of the global boundary variable. If it points above the boundary the process is terminated.

Stack Shield is available for download at http://www.angelfire.com/sk/stackshield/

5.3 Stack Guard
Stack Guard is a compiler approach for defending programs and systems against "stack smashing" attacks. Stack Guard does not prevent the return address from being overwritten, instead it tries to detect when it happens and take the appropriate action (terminating the program before any damage is done).

Stack Guard accomplishes this in an ingenious way. Whenever a function is called, code is added for pushing a small value, called a "canary" value, to the stack. This value thus ends up between the local variables and the return address.

```
<table>
<thead>
<tr>
<th>Locals</th>
<th>Canary</th>
<th>Return address</th>
<th>Parameters</th>
</tr>
</thead>
</table>
```

When the function exits it checks that the canary value has not been modified before returning. The idea is that a buffer overflow in one of the local variables cannot overwrite the return address without simultaneously destroying the integrity of the canary value. It thus becomes possible to detect whether a buffer overflow has occurred before the function returns.

For this to work, the attacker must not be able to guess the canary value. If the attacker can correctly guess the canary value, he can overwrite the return address without being detected. Stack Guard can set the canary value in one of three possible ways [6]:

- Random canaries: the canary word value is chosen at random at the time the program executes. Thus the attacker cannot learn the canary value prior to the program start by searching the executable image.
- Null canary: the canary word is "null", i.e. 0x00000000. Since most string operations that are exploited by stack smashing attacks terminate on null, the attacker cannot easily spoof a series of nulls into the middle of the string.
- Terminator canary: not all string operations are terminated by null, e.g. gets() terminates on new line or end-of-file (represented as -1). The terminator canary is a combination of Null, CR, LF, and -1 (0xFF), which should terminate most string operations.

Stack Guard is available for download at http://www.immunix.org/

5.4 Stack-smashing Protector
IBM's stack-smashing protector (SSP), originally named ProPolice, is a variation of Stack Guard's approach. Like Stack Guard, SSP uses a modified compiler (GCC) to insert a canary in function calls to detect stack overflows at compilation time. The basic idea of buffer overflow detection comes from the Stack Guard system. However, it adds some interesting twists to the basic idea. The novel features are (1) the reordering of local variables to place buffers after pointers to avoid the corruption of pointers that could be used to further corrupt arbitrary memory locations, (2) the copying of pointers in function arguments to an area preceding local variable buffers to prevent the corruption of pointers that could be used to further corrupt arbitrary memory locations, and the (3) omission of instrumentation code from some functions to decrease the performance overhead [7].

SSP is available for download at http://www.trl.ibm.com/projects/security/ssp/

6. PERFORMANCE
In this section we will try to compute the performance overhead that these tools have on Linux, Pentium III, 1GHz and 256 MB RAM machines. The execution time is measured before and after the tool is applied. The performance overhead is computed as a ratio of the CPU time of a guarded function call per the base cost of the function call [13].

6.1 Micro benchmarks
We have decided to use programs based on a very simple scheme for our tests. They consist in a loop calling a function that is considered unsafe in the C/C++ language, such as strcpy(). Here is a pseudo-code illustration:

```
for (i = 0; I < N; i++)
    vulnerable_function(destination_buffer, source_buffer);
```

We ran the test program 50,000,000 times, and added the execution lengths. The time is given in seconds.

<table>
<thead>
<tr>
<th>Function</th>
<th>Exec Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>4.67</td>
</tr>
<tr>
<td>SSP</td>
<td>5.05</td>
</tr>
<tr>
<td>Libsafe</td>
<td>4.9</td>
</tr>
<tr>
<td>Stack Guard</td>
<td>5.3</td>
</tr>
<tr>
<td>Stack Shield</td>
<td>5.45</td>
</tr>
</tbody>
</table>
6.2 Macro benchmarks

In this section we will measure the overhead of execution time for these tools against various applications. Figure 3 shows the run-time cost of three applications to compare the overhead of Stack Shield, Stack Guard, Stack-Smashing Protector and Libsafe. The applications are perlbench (a CPU-bound program) measuring the time of several operations, ctags (an I/O-bound program) indexing egcs-1.1.2 directory, and imapd transmitting 100 email messages of size two kilobytes 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.

![Figure 2: Average Execution Time](image)

Table 2: Comparison of performance overhead

<table>
<thead>
<tr>
<th></th>
<th>Perl</th>
<th>Ctags</th>
<th>imapd</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSP</td>
<td>4%</td>
<td>1%</td>
<td>0%</td>
</tr>
<tr>
<td>Libsafe</td>
<td>8%</td>
<td>2%</td>
<td>0%</td>
</tr>
<tr>
<td>Stack Guard</td>
<td>8%</td>
<td>3%</td>
<td>0%</td>
</tr>
<tr>
<td>Stack Shield</td>
<td>8.5%</td>
<td>4%</td>
<td>0%</td>
</tr>
</tbody>
</table>

7. CONCLUSION

The four tools that we present in this paper are divided into two categories: static and dynamic protection methods. The three tools: SSP, Stack Guard, Stack Shield take the dynamic approach. These tools are implemented as an intermediate language translator for GCC, which means the implementation is independent of the operating systems and the processors used. However the major drawback of these three tools is that they require the dependent applications to be recompiled. It is time consuming and very costly to recompile a large system.

On the other hand, Libsafe takes the static approach that works with executables by intercepting and replacing vulnerable C library functions with safe implementations and, as such, does not require recompilation of source code, making Libsafe more applicable to legacy code. Especially in an active environment that requires a production roll out for each newly compiled application.

In respect to runtime performance, Figure 4 showed SSP has the best performance overhead in both Perlbench and Ctags. Whereas Libsafe, Stack Guard and Stack Shield come very close together with 8% - 8.5% for their runtime performance overhead. Again, these numbers are varied base on the total number of functions in a particular application and the number of string functions used in that application. Over all, the performance overhead seems to be insignificant compare to the protection that these tools provide.

8. REFERENCES


