Advances in Computer Science and Information Technology (ACSIT)
Print ISSN: 2393-9907; Online ISSN: 2393-9915; Volume 2, Number 8; April-June, 2015 pp. 41-46

http://www.krishisanskriti.org/acsit.html

Buffer Overflow Exploit Prevention

Kushal Ahuja[1] and Vinod Kumar[2]

[1,2]Department of Computer Science Delhi Technological University New Delhi, India
E-mail: [1]kushalahuja@gmail.com, [2]vinodkumar@dce.edu

Abstract—*Buffer overflow continues to be one of the leading vulnerabilities that plague the software industry. Buffer overflow as the name suggests results because software may potentially allow operation, such as reading or writing, to be performed at addresses not intended by the developer. Buffer overflow typically affects unsafe languages such as C and C++ as these languages don't perform bound checks on arrays and pointer references and they focus more on programming efficiency and code length than on the security aspects. Languages like Java that perform bound check are not prone to buffer overflows arising out of unbounded copy. Range of possible buffer overflow exploits is based on degree of control by attacker achieved. It may range from "Denial of Service" attack (resulting in system crash) to "Arbitrary Code Execution" (to hijack control of your system).*

In this report, we typically explore the kinds of programming vulnerabilities which result into Buffer Overflow, how an attacker could exploit them, how a best programmer could detect them and inhibit or prevent exploitation of those vulnerabilities. This report details one method which is based on "Execution Space Protection" to prevent buffer overflow vulnerability from being exploited. To understand this method, report is complemented with basic Process Memory Layout Details, Linux Internals like system calls, system call table, Interrupt Descriptor table (IDT), Virtual memory area, and Basic Kernel Module Programming. With all this knowledge simulated, we'll go through design details of a Kernel Module to protect buffer overflow vulnerability from being exploited. This kernel module works by overwriting the system call table function pointers with its own function. Doing so would direct the control to the module function whenever a system call is made and we can do the necessary processing to know whether system call originated from writable region of memory. If so, we can kill the system call without letting it hijack control of our system.

1. INTRODUCTION

Buffer Overflow – The software may potentially allow operations, such as reading and writing at addresses not intended by the developer.

Common Weakness Enumeration (http://cwe.mitre.org)

In computer security and programming, a buffer overflow, or buffer overrun, is an anomalous condition where a process attempts to store data beyond the boundaries of a fixed-length buffer. The result is that the extra data overwrites adjacent memory locations. The overwritten data may include other buffers, variables and program flow data, and may result in erratic program behavior, a memory access exception, program termination (a crash), incorrect results or — especially if deliberately caused by a malicious user — a possible breach of system security. A buffer overflow occurs when data written to a buffer, due to insufficient bounds checking, corrupts data values in memory addresses adjacent to the allocated buffer. Most commonly this occurs when copying strings of characters from one buffer to another.

Example:

Suppose, a program has defined two variables items which are adjacent in memory: an 8-byte-long string buffer, A, and a 2-byte integer, B. Initially, A contains nothing but zero bytes, and B contains the number length of buffer A i.e. 8.

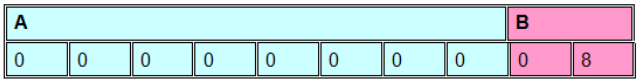


Figure 1.1: 'A' and 'B', Two Adjacent Memory Locations

Now, the program attempts to store the character string "excessive" in the A buffer, followed by a zero byte to mark the end of the string. By not checking the length of the string, it overwrites the value of B:

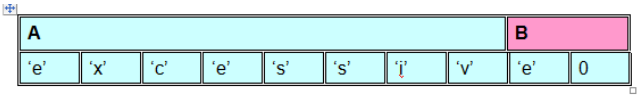


Figure 1.2: 'B' Overwritten by Unsafe Buffer Copy on 'A'

In nearly all computer languages, both old and new, trying to overflow a buffer is normally detected and prevented automatically by the language itself (say, by raising an exception or adding more space to the buffer as needed). But there are two languages where this is not true: C and C++. Often C and C++ will simply let additional data be scribbled all over the rest of the memory, and this can be exploited to horrific effect. What's worse, it's actually more difficult to write correct code in C and C++ to always deal with buffer overflows; it's very easy to accidentally permit a buffer overflow. These might be irrelevant facts except that C and C++ are very widely used; for example, 86% of the lines of code in Red Hat Linux 7.1 are in either C or C++. Thus, there's a vast amount of code that's vulnerable to this problem because the implementation language fails to protect against it.

This isn't easily fixed in the C and C++ languages themselves. The problem is based on fundamental design decisions of the C language (particularly how pointers and arrays are handled in C). Since C++ is a mostly compatible superset of C, it has the same problems. Fundamentally, any time your program reads or copies data into a buffer, it needs to check that there's enough space before making the copy. An exception is if you can show it can't happen -- but often programs are changed over time that make the impossible possible.

Another problem is that C and C++ have very weak typing for integers and don't normally detect problems manipulating them. Since they require the programmer to do all the detecting of problems by hand, it's easy to manipulate numbers incorrectly in a way that's exploitable. In particular, it's often the case that you need to keep track of a buffer length, or read a length of something. But what happens if you use a signed value to store this -- can an attacker cause it to "go negative" and then later have that data interpreted as a really large positive number? When numeric values are translated between different sizes, can an attacker exploit this? Are numeric overflows exploitable? Sometimes the way integers are handled creates a vulnerability.

2. PROCESS MEMORY LAYOUT

We would discuss about how the process memory is laid out when a process is loaded into the memory. We will go in more details of a memory unit called "Stack" and develop our understanding on Stack Frame by having some discussion about assembly equivalent of C code.

Typical Memory Layout

When a program is executed, its various compilation units are mapped in memory in a well-structured manner. The kernel arranges pages into blocks that share certain properties, such as access permissions. These blocks are called memory regions, segments, or mappings. Figure below reflects typical memory layout.

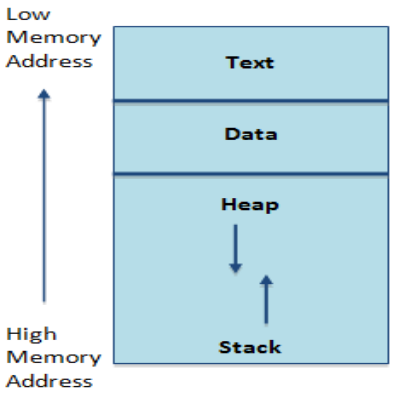


Figure 2.1: Typical Memory Layout

Text Segment - contains primarily the program code, i.e., a series of executable program instructions. Other than program code, it contains string literals, constant variables, and other read-only data. The code execution is non-linear, it can skip code, jump, and call functions on certain conditions. Therefore, we have a pointer called EIP, or instruction pointer. The address where EIP points to always contains the code that will be executed next.

Data Segment - is an area of memory containing both initialized and uninitialized global data. Its size is provided at compilation time.

Stack Segment - is used to pass data (arguments) to functions and as a space for variables of functions. It is shared by the stack (which is a LIFO data structure) and heap that, in turn, is allocated at run time. The stack is used to store function call-by arguments, local variables and values of selected registers allowing it to retrieve the program state. The heap holds dynamic variables. To allocate memory, the heap uses the malloc function or the new operator.

Stack Frame

Let us try to explore more about the stack area shown in Fig. 2.1, with the help of a sample program when compiled using GNU compiler:

```
main() {
      int i, j, k;
      i = 400
      j = 500
      k = add(i+j)
}
int add(int a, int b) {
      int c;
      c = a+b;
}
```

Compiler takes source code and emits assembly code. The following steps are involved in compilation of the above code into GNU assembly equivalent:

1. Identify executable and non-executable statements within the source code.
2. Construct a local variable table and resolve all non-executable statements.
3. Convert executable statements into assembly equivalents as per GNU assembly template.

In step (1), when we look up the source code, we'll find two kinds of statements:

1) Executable - which need CPU time
2) Non-executable - which don't need CPU time. These statements like local variable declarations find their place on stack. Stack is a LIFO structure.

In step (2), we create a table for non-executable statements called local variables table or symbol table. Columns of this table are Symbol Name, Type, Composition(memory space needed), address. Every function has its own symbol table. For above sample code's main() function, symbol table appears like:

Table 2.1: Function Symbol Table

Symbol Name	Type	Composition	Address
i	int	4	-12 (%ebp)
j	int	4	-8 (%ebp)
k	int	4	-4 (%ebp)
Total		**12**	

Extended Base Pointer (ebp) and Extended Stack Pointer (esp), are the CPU registers that are referenced throughout the assembly code. EBP contains virtual address, the address at compile time. On top of EBP, resides the stack. Local variables and arguments are addressed with respect to EBP. Arguments stay in the high memory region and local variables in low memory region with respect to EBP. Therefore, EBP is also known as Stack Frame Pointer. For the sample code, first variable is at -4 offset, second at -8 and third at -12 offset from EBP as stack grows upwards.

In step (3), we have to convert executable code into assembly equivalent. Template for assembly equivalent looks something like below:

```
Function add():
    Prologue;
    Function body;
    Epilogue;
```

Prologue (denotes opening brace '{') and epilogue (denotes closing brace '}') are fixed for every function.

```
Prologue:
    pushl    %ebp          } 'enter' instruction on x86
    movl %esp, %ebp        }
Epilogue:
    movl   %ebp,   %esp    }
    popl           %ebp    }
                           } 'leave' instruction on x86
    ret
```

Function call to "add" ('call' instruction in x86) would push the arguments on the stack typically from right to left. Then the Return Address from where 'main' will resume its execution after 'add' returns, is pushed on to the stack. Then the function prologue gets executed and the stack is allocated for the local variables of 'add'. When 'add' finishes its execution, epilogue, which is just reverse of prologue, gets executed and 'ret' instruction pops back the return address from stack for 'main' to resume. So the stack frame for 'add' appears like:

3. BUFFER OVERFLOW EXPLOITATION, DETECTION AND PREVENTION

3.1 Buffer Overflow Vulnerabilities

Besides unbounded methods like strcpy(), strcat(), sprintf(), gets() and memcpy() etc., which are so called the reasons of buffer overflows, there are other reasons as well which may make a program vulnerable to be exploited by the attacker. Let's discuss some few of them here.

Unbounded Transfer

Improper Termination

Buffer Underwrite

Buffer Overflow Detection

Static analysis as well as runtime analysis of the code can protect a programmer from introducing buffer overflow vulnerabilities in the production environment where the software has to be actually deployed.

Static Analysis

There are tools available like Klockwork and Coverity which contain rules to check for secure coding violations. CiscoProduct Security Group evaluated both the static analysis tools against violations detailed in CERT's secure coding guidelines and ISO safe C technical doc and observed that Coverity had lower false positive rates, more detailed and intelligent messages, and was able to detect elusive bugs that span multiple functions.

Compile and Runtime Analysis

Compilers options GCC 4.0+ and -D_FORTIFY_SOURCE=1/2, are provided which can perform light weight checks to detect common buffer overflows. These options may infact warn at compile time if hey could detect potential buffer overflow at compile time and replace copy functions (variants of memcpy, strcpy, strat, sprintf, gest etc.) with runtime checking versions which take the length of the destination object. These compiler options enables a programmer to abort the program if overflow is detected at runtime.

Buffer Overflow Protection

Of course, it's hard to get programmers to not make common mistakes, and it's often difficult to change programs (and programmers!) to another language. So why not have the underlying system automatically protect against these problems? At the very least, protection against stack-smashing attacks would be a good thing, because stack-smashing attacks are especially easy to do.

In general, changing the underlying system so that it protects against common security problems is an excellent idea, and we'll encounter that theme in later articles too. It turns out there are many defensive measures available, and some of the most popular measures can be grouped into these categories:

- Canary-based defenses. This includes StackGuard (as used by Immunix), ssp/ProPolice (as used by OpenBSD), and Microsoft's /GS option.

- Non-executing stack defenses. This includes Solar Designer's non-exec stack patch (as used by OpenWall) and exec shield (as used by Red Hat/Fedora).
- Other approaches. This includes libsafe (as used by Mandrake) and split-stack approaches.

4. KERNEL MODULE PROGRAMMING

One of the good features of Linux is the ability to extend at runtime the set of features offered by the kernel. This means that we can add functionality to the kernel (and remove functionality as well) while the system is up and running without needing to reboot the system. Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types(or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program. These are special shared objects (like .so files) having extension '.ko' where 'k' indicates that it is a kernel object. Kernel modules are nothing but C files which can be loaded and unloaded into the kernel upon demand. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image.

4.1 Writing a Kernel Module
4.2 Compiling and Building Modules
4.3 Loading and Unloading Modules

5. FUNCTIONAL SPECIFICATIONS FOR BOEP

Salient feature of the our kernel module implementation is that it would allow buffer overflow exploit to write beyond the bounds of a program buffer but it would prevent it from impairing our system security.

Below is the general set of features, the module should provide:

1. Locate the system call table, save current state of it and overwrite it with our own function pointer at load-time.
2. Whenever a system call is invoked from user space, the control should be passed to the module function with which we replaced the system call table.
3. The module function should ensure that if the system call originated from the writable region of memory, it should be killed and if not, control should be given back to the actual system call service routine.
4. On unloading, the system call table should be restored to its original state.

Let's represent what we said above, in a form of a flowchart to help us understand the program flow:

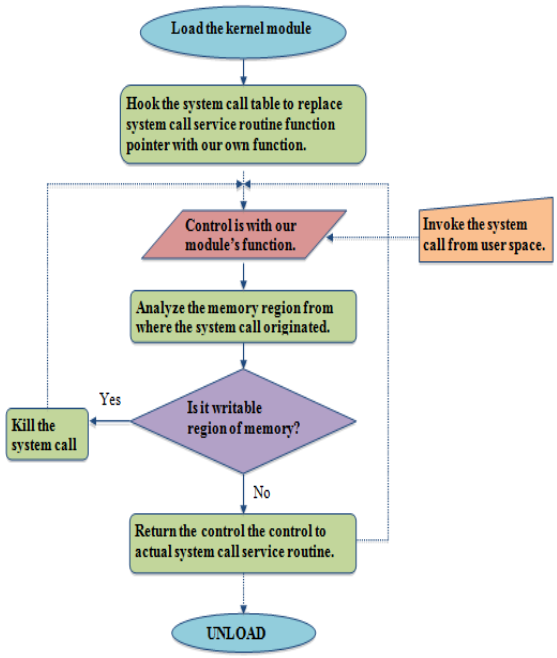


Figure 5.1: BOEP execution flow chart

6. HOOKING SYSTEM CALL TABLE

This module should generically detect and prevent buffer overflow attacks on Linux by determining if a system call originated from a writable region of memory. If so, it kills the system call. Doing it, requires knowing the system call table address, so that we may hook the table to point to our module function. So, we should be able to locate sys_call_table without the exported symbol. Till linux 2.4 kernel, there was an exported symbol that could give the system call table address like:

From linux 2.6 onwards, this functionality was removed for three primary reasons:

1. It made it too easy for a programmer to accidentally trash the entire system with a single module.
2. It made it too easy for a programmer to subvert the entire system, including security etc. with a single module.
3. It was felt that the existing kernel functions where more than adequate for normal module programming.

Before going any further, we need to have some basic knowledge about the what is a system call, system call table because kernel module implementation would require overwriting the function pointers in system call table with our module function.

System Call

The role of kernel is to collect the requirements from user and to run the application by providing them kernel resources. Kernel abstracts the application from all hardware issues like resource management etc. This communication from user-

space application to kernel-space is made possible through system calls. So, system calls are kernel space functions that serve as an interface between kernel and the applications.

A unique number identifies each system call in linux kernel. To see it, go to kernel source directory say ~/kernel-2.6. In file include/asm-i386/unistd.h under kernel source tree, we'll find a list of system calls and corresponding identifiers. The identifiers start with 0 and run through some finite number 293 or so. An example entry for "read" system call in unistd.h is:

The macro NR_syscalls contains the total number of system calls for a kernel.

System Call Handler and Service Routine

As discussed above, When a User Mode process invokes a system call, the CPU switches to Kernel Mode and starts the execution of a kernel function. The result is a jump to an assembly language function called the system call handler. Because the kernel implements many different system calls, the User Mode process must pass a parameter called the system call number to identify the required system call; the eax register is used by Linux for this purpose. General execution flow while invoking a system call is:

- The system call number as seen from unistd.h, is loaded into eax.
- All the parameters for system call are pushed into CPU registers. But to pass the parameters in registers, two conditions must be satisfied:
  1. The length of each parameter cannot exceed the length of a register (32 bits for 32-bit architecture).
  2. The number of parameters must not exceed six, besides the system call number passed in eax, because 80 x 86 processors have a very limited number of registers.

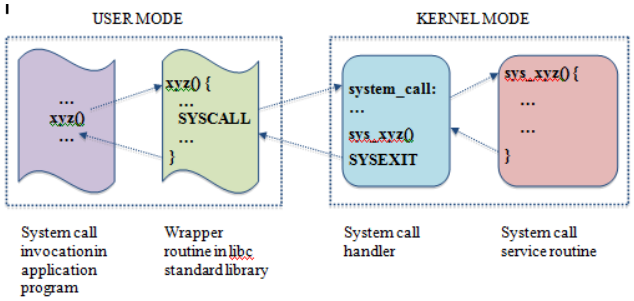
However, system calls that require more than six parameters exist. In such cases, a single register is used to point to a memory area in the process address space that contains the parameter values. Of course, programmers do not have to care about this workaround. As with every C function call, parameters are automatically saved on the stack when the wrapper routine is invoked. This routine will find the appropriate way to pass the parameters to the kernel.

- Now, as the signature is ready, int 0x80 is invoked to switch from user-space to kernel-space.
- In kernel-space, eax is read back to see the service routine to be executed to serve the user-space system call.

The system call handler, which has a structure similar to that of the other exception handlers, performs the following operations:

- Saves the contents of most registers in the Kernel Mode stack (this operation is common to all system calls and is coded in assembly language).
- Handles the system call by invoking a corresponding C function called the system call service routine.
- Exits from the handler: the registers are loaded with the values saved in the Kernel Mode stack, and the CPU is switched back from Kernel Mode to User Mode (this operation is common to all system calls and is coded in assembly language).

The name of the service routine associated with the xyz( ) system call is usually sys_xyz(); there are, however, a few exceptions to this rule. Figure below explains the execution described above.



Protection System Call Handler

After hooking the system call table, core stuff left is to know, whether the memory region from where the system call is launched is writable or not. Because generally processor doesn't allow to execute from writable region of memory, we'll kill the system.

7. SUMMARY, CONCLUSION AND FUTURE WORK

Summary

As part of this report, we learnt about what are buffer overflow vulnerabilities, how they are exploited, and how we can prevent them from being exploited. We dived into the linux kernel and learnt its inner workings. Then, we used kernel module programming to prevent the buffer overflow attacks from hijacking our systems.

Conclusion

Buffer overflow vulnerabilities can be minimized by following secure coding practices but still it's difficult to completely eradicate them because to err is human. Moreover, it's a tedious job to sit and scrutinize all the existing applications for possible buffer overflow vulnerabilities. This is where the Buffer Overflow Exploit Prevention module becomes useful; you can simply load it at run-time in your kernel and just forget about your system security, which can otherwise be impaired by an attacker**.**

Future Work

The future work in this direction can involve:

1. Implementing an ioctl interface to interact with the kernel module from user space to pass it on a set of set of system calls and corresponding actions like kill or ignore.
2. Expanded reporting to the user space.
3. Finding a way to deal with the "system calls" that come from kernel.

REFERENCES

[1] Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman. Linux Device Drivers. Sebastopol, USA: O'REILLY, Third Edition, 2005

[2] Robert Love, Linux Kernel Development. Indianapolis, USA: Pearson Education, Second Edition, 2005

[3] Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel. Sebastopol, USA: O'Reilly, Third Edition, 2005

[4] John Qian. "How to Kill Buffer Overflows – What is it". Product Security Workshop. <http://wwwin-enged.cisco.com/etools/videolibrary_public/cgi-bin/>

[5] David A. Wheeler. "Countering Buffer Overflows". Secure Programmer. <http://www.ibm.com/developerworks/linux/library/l-sp4.html>

[6] Common Weakness Enumeration. <http://cwe.mitre.org/>

[7] Analysis of Buffer Overflow Attacks. <http://www.windowsecurity.com/articles/Analysis_of_Buffer_Overflow_Attacks.html>

[8] Istvan Simon. "A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks". <http://www.mcs.csuhayward.edu/~simon/security/boflo.html>

[9]