PrabhakerMateti

January 29, 2011

1 Buffer Overflow

1.1 Prabhaker Mateti

Abstract: A large number of exploits have been due to sloppy software development. Exceeding array bounds is referred to in security circles as "buffer overflow." These are by far the most common security problems in software. This lecture explains the stack-smashing technique, and presents a few techniques that help in avoiding the exploit.

Slides — Extracted Code from *AlephOne* — This article is part of Internet Security Lectures

pmNotes-BufOvflo-20080507-acer602.txt These are the results of trying out the code examples from the AlephOne article. These notes were recorded with Auditor LiveCD on my old Acer laptop with Pentium III (Coppermine) running Debian GNU/Linux 3.1, gcc version 3.3.5 (Debian 1:3.3.5-2).

1.2 Table of Contents

1.3 Educational Objective

s

A large number of exploits have been due to sloppy software development. A surprisingly large percentage of these are attributable to exceeding array bounds, that is referred to in security circles as "buffer overflow."

1. Present several real life examples of buffer overflow.
2. Describe the stack smashing technique
3. Describe several techniques of overflow exploit avoidance.

2 Buffer Overflow

Buffer overflow is a common programming error. There is not a single OS that is free from this error. Buffer overflows have been causing serious security problems for decades. In the most famous example, the Internet worm of 1988 used a buffer overflow in `fingerd`. What is surprising is that a number of security oriented software such as SSH and Kerberos also have these errors.

2.1 The Buffer Overflow Error

The essence of this problem can be explained by the following. The line `strcpy(p, q)` is a common piece of code in most systems programs. An example of this is: `char env[32]; strcpy(env, getenv("TERM"));` The `strcpy(p, q)` is proper only when

1. p is pointing to a char array of size m,
2. q is pointing to a char array of size n,
3. m ¿= n,
4. q[i] == '\0' for some i where `0 <= i <= n-1`

Unfortunately, only a few programs verify that all the above hold prior to invoking `strcpy(p, q)`. A buffer overflow occurs when an object of size `m + d` is placed into a container of size `m`. This can happen in many situations when the programmer does not take proper care to bounds check what their functions do and what they are placing into variables inside their programs. If `n > m` in the `strcpy(p, q)` of above an area of memory beyond `&p[m]` gets overwritten.

A few other examples of such buffer overflows:

- `char input[20]; gets(input);`
- memcpy(p, q);

2.2 Buffer Overflow Exploits

An attacker exploits this programming mistake. He injects cleverly constructed data / executable-code into the area beyond the declared sizes. If the "buffer" is a local C variable, the overflow can be used to force the function to run code of an attackers' choosing. This specific variation is often called a "stack smashing" attack. A buffer in the heap isn't much better. Attackers have been able to use such overflows to control other variables in the program.

2.2.1 Stack Smashing

Stack-smashing attacks target a specific programming fault: careless use of variables allocated on the program's run-time stack such as local variables and function arguments. The idea is straightforward: Insert attack code (for example, code that invokes a shell) somewhere and overwrite the stack in such a way that control gets passed to the attack code. If the program being exploited runs with root privilege, the attacker gets that privilege in the interactive session.

The paper by Aleph One, "Smashing The Stack For Fun And Profit," describes the technique in great detail, and is required reading. But it has a few inaccuracies. A version of this paper with a few of my corrections in place is here. A few additional comments are included below.

Buffer overflow code intimately depends on (i) the CPU, (ii) the OS and (iii) the compiler of the language that the code is in. Aleph One's is paper is written in 1990s, and the examples are based on a Linux version running on an x86 32-bit machine. If you are trying his code, on a machine today, you need to adjust it. Most Linux installations now use libraries that can detect stack smashing.

2.2.2 Heap overflows versus stack overflows

"Heap overflows are generally much harder to exploit than stack overflows (although successful heap overflow attacks do exist). For this reason, some programmers never statically allocate buffers. Instead, they malloc() or new everything, and believe this will protect them from overflow problems. Often they are right, because there aren't many people who have the expertise required to exploit heap overflows. But dynamic buffer allocation is not intrinsically less dangerous than other approaches. Don't rely on dynamic allocation for everything and forget about the buffer overflow problem. Dynamic allocation is not a cure-all."

For more details on heap overflows, read the article "w00w00 on Heap Overflows" cited in the references.

2.3 Techniques of Avoiding Buffer Overflow

2.3.1 Modern Programming Languages

Most modern programming languages are essentially immune to this problem, either because they automatically resize arrays (e.g., Perl, and Java), or because they normally detect and prevent buffer overflows (e.g., Ada95 and Java). However, the C language provides no protection against such problems, and C++ can be easily used in ways to cause this problem too.

2.3.2 Careful Use of C/C++ Library Functions

C users must avoid using functions that do not check bounds unless they've ensured the bounds will never get exceeded. Functions to avoid in most cases include: `strcpy(3), strcat(3), sprintf(3), and gets(3)`. These should be replaced with functions such as `strncpy(3), strncat(3), snprintf(3), and fgets(3)` respectively, but see the discussion below. The function `strlen(3)` should be avoided unless you can guarantee that there will be a terminating NUL (ascii code zero) character to find. Other functions that may permit buffer overruns include `fscanf(3), scanf(3), vsprintf(3), realpath(3), getopt(3), getpass(3), streadd(3), strecpy(3),` and `strtrns(3)`.

Beware that calls to `strncpy(3)` and `strncat(3)` have somewhat surprising semantics and are hard to use correctly. E.g., the function strncpy(3) does not NUL-terminate the destination string if the source string length is at least equal to the destination's, so be sure to set the last character of the destination string to NUL after calling `strncpy(3)`. If you're going to reuse the same buffer many times, an efficient approach is to tell `strncpy()` that the buffer is one character shorter than it actually is and set the last character to NUL once before use. Both `strncpy(3)` and `strncat(3)` require that you pass the amount of space available. Neither provide a simple mechanism to determine *if* an overflow has occurred. Finally, `strncpy(3)` has a significant performance penalty compared to the `strcpy(3)`, because `strncpy(3)` NUL-fills the remainder of the destination.

2.3.3 Static and Dynamically Allocated Buffers

The fact that a buffer is a fixed length may be exploitable. The basic idea is that the attacker sets up a really long string so that, when the string is truncated, the final result will be what the attacker wanted (instead of what the developer intended). Perhaps the string is catenated from several smaller pieces; the attacker might make the first piece as long as the entire buffer, so all later attempts to concatenate strings do nothing. Here are some specific examples:

- Imagine code that calls `gethostbyname(3)` and, if successful, immediately copies `hostent->h_name` to a fixed-size buffer using strncpy or snprintf. Using strncpy or snprintf protects against an overflow of an excessively long fully-qualified domain name (FQDN), so you might think you're done.

However, this could result in chopping off the end of the FQDN. This may be very undesirable, depending on what happens next.

- Imagine code that uses `strncpy, strncat, snprintf,` etc., to copy the full path of a filesystem object to some buffer. Further imagine that the original value was provided by an untrusted user, and that the copying is part of a process to pass a resulting computation to a function. Now imagine that an attacker pads a path with a large number of '/'s at the beginning. This could result in future operations being performed on the file "/". If the program appends values in the belief that the result will be safe, the program may be exploitable. Or, the attacker could devise a long filename near the buffer length, so that attempts to append to the filename would silently fail.

An alternative is to dynamically reallocate all strings instead of using fixed-size buffers. This general approach is recommended by the GNU programming guidelines, since it permits programs to handle arbitrarily-sized inputs (until they run out of memory). The memory may even be exhausted at some other point in the program than the portion where you're worried about buffer overflows; any memory allocation can fail. Also, since dynamic reallocation may cause memory to be inefficiently allocated, it is entirely possible to run out of memory even though technically there is enough virtual memory available to the program to continue. In addition, before running out of memory the program will probably use a great deal of virtual memory; this can easily result in "thrashing", a situation in which the computer spends all its time just shuttling information between the disk and memory (instead of doing useful work). This can have the effect of a denial of service attack. Some rational limits on input size can help here. In general, the program must be designed to be fail safe when memory is exhausted.

2.3.4 Newer Libraries

Newer libraries for C include the `strlcpy(3)` and `strlcat(3)` functions, with prototypes:

```
    size_t strlcpy (char *dst, const char *src, size_t size);
size_t strlcat (char *dst, const char *src, size_t size);
```

Both `strlcpy` and `strlcat` take the full size of the destination buffer as a parameter (not the maximum number of characters to be copied) and guarantee to NUL-terminate the result (as long as size is larger than 0). The `strlcpy` function copies up to `size-1` characters from the NUL-terminated string `src` to `dst`, NUL-terminating the result. The `strlcat` function appends the NUL-terminated string `src` to the end of `dst`. It will append at most `size - strlen(dst) - 1` bytes, NUL-terminating the result.

One nuisance is that these two functions are not, by default, installed in most Unix-like systems.

2.3.5 Compilation Solutions in C/C++

Newer compilers perform bounds-checking. Visit, e.g., `http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.` html . Such tools provide one more layer of defense, but it's not wise to depend on this technique as your sole defense. There are at least two reasons for this. First, most such tools only provide partial defense against buffer overflows (and the "complete" defenses are generally 10-30 times slower); C and C++ were simply not designed to protect against buffer overflow. Second, for open source programs you cannot be certain what tools will be used to compile the program; using the default "normal" compiler for a given system might suddenly open security flaws.

StackGuard [Cowan et al. 1998] is a modification of the standard GNU C compiler gcc. StackGuard works by inserting a "guard" value (called a "canary", as in how this bird was used in mines) in front of the return address; if a buffer overflow overwrites the return address, the canary's value (hopefully) changes and the system detects this before using it. This is quite valuable, but note that this does not protect against buffer overflows overwriting other values (which they may still be able to use to attack a system).

2.3.6 Non-executable user stack area

It is possible to modify the Linux kernel so that the stack segment is not executable (see grsecurity). However, this is not built into the standard Linux 2.4 kernels. In Linux 2.6.x kernels, you must chose grsecurity configuration options. Part of the rationale is that this is less protection than it appears; attackers can simply force the system to call other "interesting" locations already in the program (e.g., in its library, the heap, or static data segments). Also, sometimes Linux does require executable code in the stack, e.g., to implement signals and to implement GCC "trampolines".

Even in the presence of non-executable stack, Linux Torvalds explains that "It's _really_ easy. You do something like this: 1) overflow the buffer on the stack, so that the return value is overwritten by a pointer to the `system()` library function. 2) the next four bytes are crap (a "return pointer" for the system call, which you don't care about) 3) the next four bytes are a pointer to some random place in the shared library again that contains the string `"/bin/sh"` (and yes, just do a `strings` on the thing and you'll find it). Voila. You didn't have to write any code, the _only_ thing you needed to know was where the library is loaded by default. And yes, it's library-specific, but hey, you just select one specific commonly used version to crash. Suddenly you have a root shell on the system. So it's not only doable, it's fairly trivial to do. In short, anybody who thinks that the non-executable stack gives them any real security is very very much living in a dream world. It may catch a few attacks for old binaries that have security problems, but the basic problem is that the binaries allow you to overwrite their stacks. And if they allow that, then they allow the above exploit. It probably takes all of five lines of changes to some existing exploit, and some random program to find out where in the address space the shared libraries tend to be loaded."

In short, it's better to work first on developing a correct program that defends itself against buffer overflows. Then, after you've done this, by all means use techniques and tools like StackGuard as an additional safety net. If you've worked hard to eliminate buffer overflows in the code itself, then StackGuard is likely to be more effective because there will be fewer "chinks in the armor" that StackGuard will be called on to protect.

2.3.7 No set-user-id Programs

An attacker targets set user id programs so that after the exploit he is the root, and can do arbitrary things. So, some "people believe that if their program is not running suid root, they don't have to worry about security problems in their code, since the program can't be leveraged to achieve greater access levels. That idea has some merit, but is still a risky proposition. For one thing, you never know who is going to take your program and set the suid bit on the binary. When people can't get something to work properly, they get desperate. We've seen this sort of situation lead to entire directories of programs needlessly set setuid root."

"There can also be users of your software with no privileges at all. That means any successful buffer overflow attack will give them more privileges than they previously had. Usually, such attacks involve the network. For example, a buffer overflow in a network server program that can be tickled by outside users may provide an attacker with a login on the machine. The resulting session has the privileges of the process running the compromised network service. This type of attack happens all the time. Often, such services run as root (and generally for no good reason other than to make use of a privileged low port). Even when such services don't run as root, as soon as a cracker gets an interactive shell on a machine, it is usually only a matter of time before the machine is "owned" -- that is, the attacker gains complete control over the machine, such as root access on a UNIX box or administrator access on a Windows NT box. Such control is typically garnered by running a different exploit through the interactive shell to escalate privileges." [Quoted from `http://www-4.ibm.com/` software/ developer/ library/ buffer-defend.html? dwzone=security]

2.4 Lab Experiment

All work should be carried out in Operating Systems and Internet Security (OSIS) Lab, 429 Russ. Use any of the PCs numbered 23 to 30. No other WSU facilities are allowed.

Objective: Understand the stack smashing buffer exploit *thoroughly.*

1. Download the article by Aleph One (see References). You will be extracting the source code of `exploit3.c` and `exploit4.c` files from it.

2. Study the code of `exploit3.c` and `exploit4.c` that you extracted.

3. Improve the code so that there are no warning messages from `gcc` even after using the flags as in
   `gcc -ansi -pedantic -Wall`.

4. Reduce the size of their compiled binaries by at least 5% as seen by the `size` command when exactly the same flags are used in the compilation. Make sure no functionality is lost. Do not just remove printf's

5. Login as a non-root user. Verify that the exploit still works on the `vulnerable` program. (It may not!)

6. Turn in a lab report (of say 2-4 pages) with answers to the questions below, and thoroughly describing your changes, and how you verified that there was no loss of functionality. Include properly indented versions of your `exploit[34].c` files. Use `indent -kr`.

7. Answer the question: What is the "environment"?

8. Answer the question: Why does `exploit3.c` run `system("/bin/bash")` at the end of `main()`?

9. Search the web and report on at least two recent (within last two years) buffer overflow attacks.

2.5 Acknowledgements

The section on "Techniques of Avoiding Buffer Overflow" is based on "Secure Programming for Linux and Unix HOWTO" and the "The Unix Secure Programming FAQ."

2.6 References

1. Aleph One, "Smashing The Stack For Fun And Profit," Phrack, Vol 7, Issue 49, File 14 of 16, www. phrack.com. A classic article. local copy (.txt) is the original paper as-is. But it has a few inaccuracies. alephOne.html is the version of this paper with my corrections in place. Required Reading.

2. Tim Werthmann and Horst Gortz, Survey on Buffer Overflow Attacks and Countermeasures, Institute for IT-Security. Ruhr-University Bochum, Germany, 2006, www.nds.rub.de/lehre/seminar/SS06/Werthmann_ Buffer-Overflow. pdf Reference.

3. Matt Conover, and WSD, "w00w00 on Heap Overflows", January 1999, www.w00w00.org/ files/ articles/heaptut.txt Reference.

4. Peter Baer Galvin, "The Unix Secure Programming FAQ: Tips on security design principles, programming methods, and testing," SunWorld Magazine, Aug 1998, packetstorm.decepticons.org/ programming-tutorials/ unix.secure.programming.html [Local Copy] Required Reading.

5. David A. Wheeler, "Secure Programming for Linux and Unix HOWTO," April 2000, tldp.org/HOWTO/Secure-Programs-HOWTO/index.html Highly recommended reading.