1 An Exploit Example, a Buffer Overflow

A buffer overflow is exactly what it sounds like, the attacker overflows a buffer in the program. Why would you want to do this? Remember that the stack is where the functions store their linkage information. In particular, the stack is where a function stores the return address of the function that called it. If a buffer is located on the stack, then an overflow may allow the attacker to overwrite the return address, which will allow the attacker to take over the program! Taking over your emacs session may not accomplish much, but taking over a program like sendmail (that runs as root) means that the hacker can effectively grant themselves root access to a remote machine!

Consider the following, and admittedly contrived, program:

```
int add(){
  int i1 = read();
  int i2 = read();
  return i1 + i2;
}

int read(){
  char str[8];
  int val;

  scanf("%s", str);

    ...
}
```

Just before line 6 is executed, the stack, in memory will look something like this:

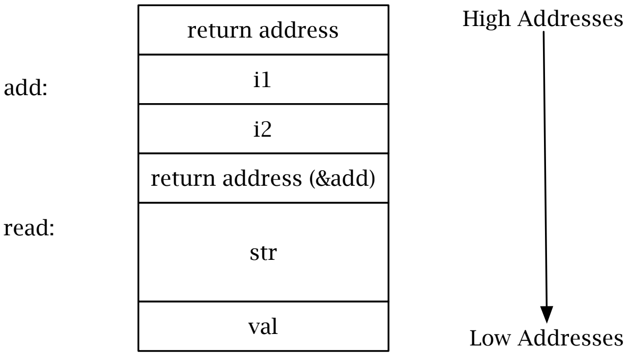

Figure 1: The stack, before an overflow

Note that `str` is a stack allocated character array, with space for only 8 characters. The call to `scanf` will copy characters from the console into `str`. This is fine as long as the user enters only 7 or fewer characters (like ”y” or ”hello”), but what if the user enters the string ”AAAAAAAAAAAA”? Then the buffer is overflowed, and the stack will look like:

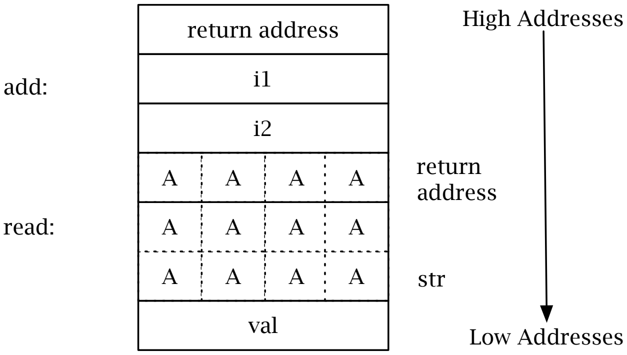

Figure 2: The stack, after an overflow

At this point, all you've done is just crash the program, however you can change the return address to anything you want. So why not overflow the buffer with your own code and just jump into that? Sure it sounds crazy, but why not? Of course, there's some logistical issues.

1. Need to calculate the return address so that it jumps into the stack
2. Need to write assembly code to do ‘something bad’

For the first challenge, you can use a tool like `gdb` to figure out where things are on the stack at any given point in the execution. Of course, because the binary is loaded dynamically there's going to be some variation. But, it turns out, not that much. This variation is combatted by selecting a return address that fits in the range of variation and then padding the beginning of your code with a whole bunch of NOPs (this is the so-called NOP sled). That way, you can return anywhere in the middle of the NOPs and it won't effect the execution of your program.

For the second challenge, we have to remember that C strings are null-terminated. The null character is encoded as the byte `0x00`, so any instruction, that when assembled contains `0x00` is out. Usually you can avoid these inconvenient instructions by replacing them with equivalent sequences of instructions without the null character. Now, what's something bad? Well, how about launching a shell as root? For example the following code is null-free shellcode to launch a bourne shell on a SPARC machine running Solaris:

```
execsh: xor %sp, %sp, %o0 ! %o0 = 0;
mov 23, %g1
ta 8 ! setuid(0);
set 0x2f62696e, %l0 ! (void*)sh = '/bin';
set 0x2f736800, %l1 ! (void*)sh + 4 = '/sh0';
sub %sp, 16, %o0 ! %o0 = '/bin/sh';
sub %sp, 8, %o1 ! %o1 = {'/bin/sh', NULL};
xor %sp, %sp, %o2 ! %o2 = NULL;
std %l0, [%sp - 16]
st %o0, [%sp - 8] ! argv[0] = sh;
st %g0, [%sp - 4] ! argv[1] = NULL;
mov 59, %g1
ta 8 ! execve(sh, argv, NULL);
xor %sp, %sp, %o0 ! %o0 = 0;
mov 1, %g1
ta 8 ! exit(0)
```

This code actually loads the literal string `"/bin/sh"` into a register (on x86 you'd put this value on the stack) and invokes `execve` to launch the program. This is accomplished with the `ta 8` instruction, which is the SPARC trap (or software interrupt) instruction.

Note that all kinds of bad things can be made to happen with shellcode. Launching a root shell is just the oldest. It is totally possible to open a network connection on a port, basically granting an intruder a hidden `rsh` connection. On windows machines, it is common to use low level APIs to fetch malicious executables over the network and install them as administrator.

Note also, that many systems are potentially immune from this kind of attack. Most hardware supports protection bits to allow/disallow reading, writing, and executing. If you just set the execute bit to 0 for stack memory, then jumping to a stack address will cause a page-fault. Interestingly, enough however, intel x86 hardware does not have an execute protection bit. So this was impossible for a while on intel Linux boxes (however, there is a kernel modification called PAX that engineers inconsistencies between the instruction and data cache to emulate an execution bit).

2 Other Software Flaws

The buffer overflow illustrated above is, strictly speaking, a stack overflow. There are many other kinds of bugs (which I get into below). These bugs all allow carefully engineered user input to modify priveledged application data. This list, which is hardly exhaustive, is here to illustrate how sophisticated and subtle these attacks can be.

2.1 Heap Overflow

A heap overflow is just like a stack overflow, except you overflow a buffer on the heap. As you can guess, this might be damaging if the overflowed buffer is next to heap data containing passwords, user ids, etc. But, even if sensitive data cannot be directly overwritten, you can still cause all kinds of damage from the heap. In fact, from the heap or BSS segment it is sometimes possible to overwrite procedure linkage information. The procedure linkage table (PLT) is a data structure used by the system linker (`ld.so`) to dynamically link code together. Essentially, the PLT provides a layer of indirection between library code and application code (this is necessary so that the system can load libraries anywhere in memory). It's just an array of addresses, basically. When application code calls a library method, it does so by basically looking up an offset in the PLT and jumping to the address contained there. If you can overwrite PLT entries, then you can redirect the control of a program anywhere you want. In this style of exploit, you provide your code somewhere on the heap and overwrite the PLT entry for a common function (`printf`, say). The next time the application invokes that function, it will, in fact be calling your malicious code.

2.2 `malloc` exploits

`malloc` is the standard way of allocating dynamic memory in C, but `malloc` itself is just another C program. `malloc` and `free` need to store information in the heap to operate correctly, which means that a carefully constructed heap-overflow can confuse `malloc` and get it to write arbitrary values to memory. This meta-data bookends normal heap allocations to allow free to know how much space to deallocate. Essentially, this meta-data is a size field, a previous pointer and a next pointer. The pointers are there because free chunks are arranged into a doubly-linked list. So, as in any doubly-linked list, when you allocate a free chunk, you need to unlink it from its next and previous neighbors. What this means is writing the next pointer to the previous chunks next pointer field (calculated from our previous pointer); and then writing our previous pointer to the next chunk's previous field (calculated from our next pointer). An exploit would either overwrite legitimate forward and back pointers, or would construct a fake free chunk that the user would then free (this actually happens!). Think what would happen if the next pointer was overwritten to be the address of some code we'd loaded into the heap and the previous pointer was the location of a return address on the stack! (or of a PLT entry, etc. etc.).

2.3 Integer Overflow

Integer overflows exploit the fact that C integers have limited precision. Values can 'roll over' from very large values to very small values. Usually, you're interested in cases where it rolls over from `0xFFFFFFFF` to 0 (or vice-versa), or from 2 billion (or so) to -1. Often, this can be used to bypass bounds checking. For instance, if the bound to be checked comes from user input, you can hand horrible values to the program (negative integers that get turned into enormous positive integers, for example). Additionally, `malloc` has a certain minimum allocation size. If a size is requested that is smaller, then `malloc` will return the minimum size. If you can induce an integer overflow so that the program ends up calling `malloc(0)`, then you'll secretly have a few bytes (usually 8) to play around with.

2.4 Exception Handler Overwriting (Windows specific)

Last is an example of an exploit that is inherent to the Windows API. Old school windows has support for an exception mechanism similar to Java's. It works by having a lookaside table that contains code pointers for various exception handlers. When the user's code installs a handler it actually just writes the function address to this table. In the event of an exception, the system (meaning the OS) runs down this table and transfers control to the appropriate exception handler. If there is no specific exception handler installed by the program, then control falls to the general exception handler, which is a pre-installed chunk of fallback code (which gently kills the program). In this exploitation strategy, you can use any technique you want to write some nasty code into program memory. Then any technique to overwrite the general exception handler (which exists in the exception table, which MUST be program writable). And lastly, you trigger an exception in the code, which will end up calling your malicious exception handler. This is the level of trickery that a true security expert must be able to manage (that's why, by the way, there are actually fairly few security experts out there).

3 Security Concepts

3.1 Protection vs. Security

Protection is ”defending or guarding against an attack”, so protection is then a set of mechanisms for controlling the access of processes and users to system resources. Security is ”freedom from damage or risk or apprehension”, security is the result of successfully enforcing an appropriate security policy. So security is basically prevention, and protection is basically damage control. More specifically, a policy is a decision about what to protect and how (e.g. laws). A mechanism is a means of implementing a policy (e.g. police). Two common mechanisms are **authorization** and **authentication**. Authorization is a mechanism that only allows permitted uses of a resource. Authentication is a mechanism to verify that a user or process is who it says it is. An important feature of security mechanisms is flexibility. Policies often change. They co-evolve with the attackers, or they are changed to fix mistakes. Therefore a mechanism that can change with an evolving policy is better than a brittle mechanism that must be replaced when a policy changes.

3.2 Reality

So, that's the ivory-tower version of security. In reality, there are many tensions and roadblocks that make effective policies difficult. First, there is often tension between **security** and **convenience**. For instance, it would be more convenient to not have to remember a username and password and log in each and every time you want to use a computer. However, the security afforded by user authentication is too beneficial to discard. More recently, Vista has been lambasted in the blogosphere for constantly prompting the user for the Administrator password to perform 'priviledged' actions. Mac OSX has employed a similar mechanism for years, however it has been streamlined so that it only pops up rarely (when changing system settings, upgrading system software, or installing software to the system drive). Vista's system, which is very similar, just pops up way more often. That increase in frequency was enough for Mac OSX to be labeled as 'secure' and Vista to be labeled as 'annoying'.

Second, security is often a brittle thing. Once it's compromised, it's often difficult (or impossible) to regain it. For example, shared secret cryptography is great as long as no one (unauthorized) gets ahold of the secret keys. If an unauthorized person were to grab them, then they could read every message ever encrypted with them. You could change your keys a dozen times over, but you wouldn't be able to prevent the compromised keys from being used on saved messages. In practice, shared key systems often generate whole sequences of keys and rotate through them fairly rapidly. This limits the amount of traffic encrypted with any one key, and thereby limits the damage (but doesn't prevent the keys from being compromised).

Third, security cannot be added as an afterthought. Microsoft had to basically re-do a whole bunch of the code for Longhorn in order to engineer security into the OS from the ground up (that's why it's now called Vista). Security policies can dictate design decisions at many levels. Security policies can impact programmer convenience, and programmers working on the project will make things as convenient as possible unless security is a major concern. Plus, programmers are often not particularly cognizent of security issues. Unless they are trained and vetted in security practices, they won't be able to easily spot security shortfalls in their code (and therefore won't be able to easily correct them).

Fourth, simple pen-testing is insufficient to gaurantee security. Much security testing takes the form of hiring security consultants to try and rip apart your software (often called penetration testing or red teaming). Of course, this is just like any other random testing in that not finding bugs doesn't mean that bugs aren't there. Interestingly enough, however, my experience has been that security consultants usually have no difficulty finding exploitable bugs.

3.3 Security Design Guidelines

The basis for a secure design is the notion of so-called 'trusted software' (aka trusted base). Each layer of the system is constructed assuming that the lower level's security gaurantees can be completely trusted. Of course this means lots of testing, but it also means that automatic verification is extremely important. For instance, type-checking a strong type system is an automatic way of proving that software lacks certain classes of errors (casting errors, array out-of-bounds errors, etc.). Combined with layered levels of trust is also the notion that each layer should be as small as possible (this is the principle of least priviledge). Namely, each layer should only be concerned with security matters pertinent to that layer. This is essentially the security analogue of a systems design truism, that each layer should hide as much information from the next layer as possible. This hiding is accomplished via abstraction through an interface.

All this trusted software runs on hardware, so the lowest software layers actually use trusted hardware mechanisms as their trusted base. For instance, OSes trust that the hardware won't allow users to switch from user to supervisor mode except through the gaurded mechanism of software interrupts (traps). Additionally, virtual memory hardware supports protection bits (usually read, write, and execute). An OS has to trust that the memory hardware will honor those protection bits. Fortunately for us, hardware engineers tend to do a better job of making trustworthy hardware than us software folks do making trustworthy software :)

Lastly is to not be naive and assume that occasional breaches will occur and design for it. One technique is the so-called 'defense in depth' where multiple systems check up on each other to make sure that nothing fishy is going on. Another technique, appropriated from redundant control systems is to have multiple independant systems that vote before taking action (disagreements can be red-flags for suspicious activity). These techniques are seldom used (they can be costly). However, most systems employ extensive logging facilities. System tools and daemons log actions in an append-only file, leaving an audit trail that system administrators can examine after a breach.

4 Security Mechanisms

4.1 Protection Domains

A protection domain is a collection of resources and priviledges. You can assign users to domains, and they will only be able to use the resources in that domain according to the priviledges assigned to that domain. This concept is often analogized to a ring of keys, giving an employee only those keys they need to do their job. In theory, one could partition the entire system into (potentially overlapping) domains. However, this is tricky with a large system, and there's the difficulty of specifying all the (resource, operation) pairs. Additionally, there's the question of implementing this in an efficient way.

4.2 Access Control Matrix

An access control matrix is a giant matrix specifying what priviledges a domain has over a set of resources. Usually, the resources form the columns, the domains the rows, and the elements are the allowed operations (priviledges). In practice, this matrix can be very sparse, so actually storing it as a matrix is probably a bad idea. If you decide to store allowed domains with the resource, you call it an **Access Control List**. If you store the allowed resources with the domains, then you call it a **Capabilities List**.

4.3 ACLs

An Access Control List (ACL) is a sparse representation of an Access Control Matrix. It's a list of allowed domains hanging off of a resource specifier. Each resource lists the domains and their rights. If the domain has no rights to this resource, it's not in the list. UNIX systems have a simplified version of the ACL embodied in the normal permissions string. There are three classes of domains: owner, group, and other. And each domain can have read, write, and/or execute priviledges. (In fact, many Unix systems also have a more fully-fledged ACL extension to standard permissions). In Unix, the owner (or root) has the right to set priviledges (modify the ACL). More generally, the right to modify the ACL is itself a priviledge that can be assigned to a domain.

4.4 Capabilities List

A capabilities list is a sparse representation of an Access Control Matrix where you have a list of resources and priviledges associated with a protection domain specifier. The domain assembles its capabilities during initialization, and these are generally protected from modification. Capabilities are stored in the kernel and are accessed via a system call, additionally specialized hardware can be used to prevent tampering of capabilities data (note that this is not so easy with ACLs). Capabilities are often protected via a form of encryption, where the capability token is a special code. Capabilities are usually considered more secure than ACLs because they're not vulnerable to the 'confused deputy problem'.

The confused deputy problem is when an underpriviledged program gains more priviledges by using an intermediary program with more priviledges. For example, a compilation server has more priviledges than a user (it needs to be able to write files anywhere). A user requests a compilation by specifying an input and an output file. If the user is 'playing nice' then the output file will be something in their home directory. But, note that there's nothing stopping a devious user from specifying a system file as an output file. In the ACL world, nothing is wrong. The file being overwritten checks its ACL and sees that the compilation server has access to do so. The devious user has just escalated their priviledges without the security infrastructure catching it. Capabilities bypasses this problem because requests must be coupled with the user's/process's capability, which can be checked by the kernel. So even though the compilation server can overwrite any file on the system, it uses the user's capability to check against. This will catch unauthorized writes.

4.5 Rings of Protection

Rings are much simpler, essentially nested concentric rings, with each inner ring adding more priviledges to the one outside it. This organization is popular because there's hardware support for it. For example, the user/supervisor mode bit is a form of ring architecture. Some processors have support for more than 2 rings (Intel supports 4).

5 Revocation

As a last resort, it may become necessary to revoke a user's/process's/system's priviledges. There are a couple of issues here, should the revocation be immediate or delayed? Should it be selective or general (e.g. can you revoke specific priviledges or just all of them)? In practice delayed revocations are simple with ACLs (you just remove the domain from the resource). Capabilities are trickier, resources may be spread out among many domains. One solution is **expiration** capabilities have a time-limit. After the time's up, you have to ask the kernel for a fresh set. This limits the time window between a decision to revoke and it taking effect. Another solution is to have **backpointers** from the resource to a list of capabilities on it. Lastly, a layer of **indirection** between the processes and the actual capabilities can be used. If the user process has to index into some kernel structure to get ahold of the actual capabilities object, then revocation can simply invalidate the entries in the kernel's table.

6 Password Exploits

Passwords in UNIX are meant to be hard to crack. Passwords are stored in the `/etc/passwd` file (or the shadow file if you're shadowing). However, they aren't stored as plaintext. This is good, it means that you can't just read the `passwd` file to get people's passwords. The actual passwords are first encrypted with a one-way function (a function that is effectively impossible to 'run backwards'), then the encrypted text is stored in `/etc/passwd`. So, when the user logs in, the text they enter for their password is fed into the one-way function. Its output is compared to the string stored in `passwd`. If it matches, then the user is authenticated. One-way functions are fairly well designed, so password crackers don't focus on trying to 'decrypt' the password strings in the password file. Instead, they usually perform a kind of informed brute-force search.

The first route of attack is to use default passwords. For instance, it is common practice to send people e-mail letting them know what their password is. Many times users don't delete these messages, nor change their passwords! If you're attacking a system where you don't have a good idea of what a password is to begin with, you can make some educated guesses. First, passwords are often short, which greatly reduces the search space (longer passwords are better). A password cracking system will usually pre-encrypt a dictionary (as well as permutations (such as camel-humps capitalization) of those words), which ends up only taking a few megs, usually. Then, the program can efficiently search for passwords that are dictionary words. If that fails, you can then resort to just generating random alpha-numeric strings and trying them. This all works very well in practice, in 1990 a study found that about 25% of passwords are guessable.

7 Intrusion Detection

Intrusion detection is exactly what it sounds like. If you can't ensure that your systems are completely secure, then you need to be able to detect when your systems are compromised (so you can contain the damage). A good intrusion detection system will detect intruders, act as a deterrent against further attacks, and provide information about attacks (for evolving defences).

In practice, intrusion detectors can work in a variety of different ways. The first is a form of pattern-matching. Some systems look for ‘bad’ patterns in network traffic, ‘bad’ patterns on the filesystem, etc. These systems usually employ a hard-coded set of rules derived from actual attacks. These systems are brittle, in the sense that a sophisticated attacker can permute their attack so that it no longer matches a built-in pattern. However, many attacks aren’t so sophisticated. Lots of attacks come from automated systems that will be caught by a pattern matching system.

Other intrusion detection systems use machine learning techniques to learn ’acceptable’ behavior and automatically derive rules for detecting ’bad’ behavior. For example, it can be possible to build up a statistical model for what kinds of traps/syscalls an application calls. An unexpected syscall would indicate an compromised process (for example, the printer daemon suddenly executing `/bin/sh`). Statistical models can be assembled for users as well (some people try to model users’ typing patterns to detect an intruder masquerading as a legitimate user). These systems theoretically offer more flexibility (i.e. less brittleness) than pattern-matching systems, however all the issues with machine learning (i.e. over-training) can be difficult to deal with. Additionally, when you train this kind of ID system, you need to make sure that you’re not training it with already compromised binaries!

In general, real intrusion detection systems employ extensive archiving in order to allow for more extensive auditing after the fact. If you’ve got the hard drive space for it, this is generally a good thing.