Chapter 2

Control-Transfer Attacks

This chapter describes a class of attacks called control-transfer attacks, in which an executing program is made to transfer control to an unintended, potentially arbitrary, piece of code. This is the most dangerous class of attacks, as it gives the successful attacker a lot of flexibility to access or change data on the compromised system.

Effecting such an attack requires the following two steps:

1. Supply a piece of malicious code (usually a “shellcode”, which spawns a shell through which the attacker can access the compromised system, often with increased privileges).

2. Cause the attacked program to transfer control to the shellcode.

The first step is usually easy: the shellcode can be stored in an environment variable, passed in the command-line argument, or fed as part of the input read by the program. One proposal for making this step more difficult is to have a non-executable stack space [Sol]. This approach has not been widely adopted because it limits the flexibility to apply certain programming techniques like “trampolines”, and because it can be circumvented with exploits such as “return-into-libc” (where control is transferred to the `system` library function to spawn a shell).

With an executable shellcode in place, the next step is to somehow make the program transfer control to the shellcode. Control-sensitive locations that may be overwritten to accomplish this include the following:
Return address: Probably the most commonly used approach, called “stack smashing” [Smi97], is to write the address of the shellcode into the return address field of the activation record. Then, when the current function returns, control is transferred to the shellcode.

Global offset table (GOT): This is a table used to dynamically resolve addresses of library functions. By writing the address of the shellcode into the GOT entry for a given library function, a subsequent call to that function will transfer control to the shellcode instead.

atexit and .dtors tables: The atexit function allows a programmer to register functions to be called when the program terminates; the address of the registered function is stored in a statically-allocated table. The .dtors section in a GNU C compiled program serves a similar purpose, storing addresses of “destructor” functions that are called when the program terminates. Writing the address of the shellcode into either of these locations will cause the shellcode to be executed when the program is about to terminate.

Function pointers: If the program has an indirect function call, the attacker could overwrite the function pointer with the address of the shellcode, so a subsequent call via that function pointer will transfer control to the shellcode.

longjmp buffers: The setjmp/longjmp mechanism is used for non-local control transfer. A call to setjmp saves information about an execution context — including the program counter — in a buffer buf such that the call longjmp(buf) will transfer control to the saved location. An attacker can overwrite the program counter in buf to cause control to be transferred to the shellcode instead.

exec/system call arguments: C library functions like system, popen, and the exec family of functions execute the command specified in their arguments. An attacker could
overwrite an argument to one of these functions with a more powerful command, like "/bin/sh".

Note that the return address, GOT, atexit table, and .dtors table are not part of the user-defined space, so a malicious write into these locations must violate memory safety. Function pointers, longjmp buffers, and library-function arguments, on the other hand, are part of the user-defined space, so they may potentially be overwritten without violating memory safety. However, their conventional usage is such that assignments to them should be very restricted, so a malicious write into one of these locations is seldom possible without an invalid write access. This observation suggests that detecting invalid write accesses is sufficient to prevent most control-transfer attacks.

The next three sections describe three known programming flaws — namely buffer overruns, format-string vulnerabilities, and erroneous frees — that can be exploited to write a malicious value into a control-sensitive location. In all three cases, exploiting the vulnerability requires an invalid write access into a control-sensitive location. An approach that detects invalid write accesses would detect an attempted exploit of these vulnerabilities, as well as other vulnerabilities that may not have been discovered. (The latter is an important property for preventing “day-zero” attacks.)

2.1 Buffer-Overrun Vulnerabilities

If a program writes a string into a buffer without checking against the size of the buffer, an attacker could supply a longer-than-expected string to cause the program to write beyond the end of the buffer. If the buffer is allocated on the stack, it is usually easy to exploit this flaw to overwrite the return address.

Figure 2.1(a) presents a piece of code with a buffer-overrun vulnerability. Column (b) gives the stack configuration at runtime, with the local variables dst, buf, and fp allocated above the return address field (‘ret’) in the function’s activation record. The function f copies a user-supplied string (pointed to by src) into a local array (buf) until a null-character is encountered. The loop does not perform bounds-checking, so an attacker can cause a buffer
overrun by supplying a `src` string longer than 16 bytes. The overrun can be exploited to overwrite the function pointer `fp` or the return address with the address of the attacker’s code.

### 2.2 Format-String Vulnerabilities

The `printf` family of functions takes as input a format string and a variable number of arguments, which it outputs to a stream by interpreting `%`-specifiers in the format string. Figure 2.2(a) gives an example usage, along with the stack configuration during the call to `printf`.

The `%d` specifier causes argument 1 (the integer 1234) to be written in string representation, while the `%n` specifier writes an integer value (the output character count) into the location pointed to by argument 2 (the address of location i). `printf` assumes that

---

1For simplicity, argument passing is assumed to occur entirely on the stack; i.e., no arguments are passed via registers.
the appropriate number of arguments have been pushed onto the stack, so it reads them off sequentially from its activation record as illustrated.

A format string with no %-specifiers will be output verbatim; as a result of this default behavior, programmers often call `printf` by feeding a user-supplied input string directly as the format-string argument to `printf`, as in Figure 2.2(b). Under normal usage, when the supplied input does not contain any %-specifiers, the program will behave as expected, and echo the supplied string. However, a malicious agent can supply an input string with %-specifiers to cause (essentially) any desired value to be written into any desired location in memory. If the input string contains a number $k$ of %-specifiers, `printf` will assume that $k+1$ arguments have been pushed onto the stack, and will interpret values on the stack as if they were supplied arguments, as shown in Figure 2.2(b). Since the `buf` array resides in the caller’s activation record (as is likely to occur in a real program), a large enough $k$ will cause `printf` to interpret the first bytes of `buf` as arguments. The attacker can thus supply the
address of a control-sensitive location in the first bytes of \texttt{buf} (the value \texttt{aabb} in the figure), followed by \textit{k} \texttt{\%}-specifiers with the last one being a \texttt{\%n}. This causes the output character count (an attacker-controllable value\footnote{Modifiers to \texttt{\%}-specifiers allow the value of the output character count to be adjusted to arbitrary values, though in a limited range. Various tricks can be used to construct a long integer value (needed to represent the address of the shellcode); see, e.g., \cite{scu01}, for details.}) to be written to the control-sensitive location whose address is \texttt{aabb}.

2.3 Erroneous Free

Heap memory management is usually implemented by storing extra information in the bytes preceding each “malloc chunk” of memory. If a program erroneously tries to free a memory block \texttt{B} that was not properly allocated or has already been freed, an attacker may be able to adjust the values around the beginning of \texttt{B} to create a bogus “malloc chunk” data structure, and trick the bookkeeping mechanism of \texttt{free} to write an arbitrary value into an arbitrary location in memory.

This section describes a specific vulnerability in the \texttt{traceroute} utility that was recently found to be exploitable \cite{Dvo00}.

2.3.1 Traceroute Vulnerability

The \texttt{traceroute} utility collects information about the path a packet takes when traveling through the internet to its destination. The \texttt{-g} option allows the user to specify up to eight gateway IP addresses, though the common usage is to specify at most one. These command-line arguments are saved using a function that \texttt{calloc}s one large buffer \texttt{B}_i and returns sub-pieces of memory from \texttt{B}_i. For example, if given two gateway arguments, the function returns \texttt{p}_1 and \texttt{p}_2 pointing to sub-pieces of \texttt{B}_i containing the two arguments, as illustrated in Figure 2.3(a). However, after each argument is processed, the program erroneously calls \texttt{free} on each sub-piece. First, the call \texttt{free(p}_1\texttt{)} unintentionally frees all of \texttt{B}_i. Next, \texttt{free(p}_2\texttt{)} results in undefined behavior, usually causing a crash.
It turns out that between the calls that free $p_1$ and $p_2$, there is a call to \texttt{calloc} that returns 16 bytes of the freed block $B_i$ for re-use. This block (which we will call $B_j$) is used to store the numerical IP address of the second gateway, converted from its string representation in the second gateway argument. This means that immediately prior to the call \texttt{free($p_2$)}, the memory is as shown in Figure 2.3(b). Immediately after the erroneous call \texttt{free($p_2$)}, there is another call that frees the \texttt{calloc}’ed block $B_j$.

This predictable behavior gives the user control of the memory surrounding the location pointed to by $p_2$. The idea behind the exploit is to adjust the values in the memory locations around $p_2$ to fool \texttt{free($p_2$)} so that it treats $p_2$ as if it points to the beginning of an allocated chunk. By creating a suitable memory layout to simulate internal data structures expected by \texttt{free}, the \texttt{free} function’s housekeeping routines can be made to write the address of the attacker’s shellcode into the Global Offset Table (GOT), which is a table used to dynamically resolve function addresses. Specifically, the exploit we tested [sor02] writes into the GOT entry for \texttt{free}, so that the final call that is supposed to free $B_j$ actually transfers control to the shellcode.

\textbf{2.3.2 GNU Memory Management}

Heap memory management is implemented in the GNU C library by storing extra information in the bytes around each “malloc chunk” of memory. The layout of the extra information for an allocated and free chunk are shown in Figure 2.4. In the allocated chunk,
Figure 2.4

*p* indicates the start of the allocated user memory. For both an allocated and a free chunk, two words of extra information — containing the previous chunk size, the current chunk size, and one bit to indicate whether the previous chunk is in use — are stored immediately preceding the user memory. Free chunks of memory are stored in a doubly-linked list, with two words of the free chunk used to store pointers to the next (*fd*) and previous (*bk*) chunks in the freelist.

Consider the memory configuration of Figure 2.5. Here we have three successive memory chunks, with *chunk*₂ in use, and *chunk*₃ free (as indicated by the *prev_inuse*₃ and *prev_inuse*₄ bits). The pointers *fd*₃ and *bk*₃ maintain *chunk*₃ in the linked list of freed chunks.

Figure 2.5
When `free(p_2)` is called, the `free` function, after recognizing that `chunk_3` is not in use (by checking `prev_inuse_4`), will consolidate `chunk_2` and `chunk_3` into one big chunk. In doing so, it first unlinks `chunk_3` from the linked list by executing the following instructions:

\[
\begin{align*}
\text{tmp\_bk} &= \text{chunk\_3}\rightarrow bk; \\
\text{tmp\_fd} &= \text{chunk\_3}\rightarrow fd; \\
\text{tmp\_fd}\rightarrow bk &= \text{tmp\_bk}; \\
\text{tmp\_bk}\rightarrow fd &= \text{tmp\_fd};
\end{align*}
\]

Notice that one of the effects of executing these unlinking instructions is that the location `fd_3\rightarrow bk` is assigned the value `bk_3`. This assignment will be used by the exploit to copy the address of the shellcode into the Global Offset Table, as explained in the next section.

### 2.3.3 The Exploit

The goal of the exploit is to create the memory layout of Figure 2.5, where `p_2` is the pointer to the second gateway argument from Figures 2.3(a) and 2.3(b), and where `chunk_3` and `chunk_4` are part of buffer `B_i`. This is illustrated in Figure 2.6, which shows how the memory layouts illustrated in Figures 2.3(b) and 2.5 line up.
The fields that must be set by the attacker are shown in bold in Figure 2.6. The attacker can set all of those fields by choosing an appropriate string for the second gateway argument. Recall that that argument is translated to the corresponding numerical IP address, which is stored in the first 16 bytes of $B_j$ (which includes the $size_2$ field). The method that does the translation matches the pattern `<number>.<number>.<number>.<number>`; the remainder of the argument string is ignored. Therefore, the $size_2$ field can be set by choosing an appropriate value for the fourth number, and the remainder of the argument string can be used to set the $size_3$, $fd_3$, $bk_3$, and $prev$-inuse $4$ fields (and to contain the shellcode). The $fd_3$ field is set to be the address of the GOT entry for free minus the offset of $bk$, and the $bk_3$ field is set to be the address of the attacker’s shellcode. Given this configuration, the unlinking done by free will write the address of the shellcode into the GOT entry for free, and the subsequent call to free $B_j$ will transfer control to the shellcode.