On the Evolution of Buffer Overflows

Matthias Vallentin

vallentin@icsi.berkeley.edu

May 20, 2007

The vast majority of software vulnerabilities still originates from buffer overflows. Many different variations of buffer overflows evolved over time, rendering them an ubiquitous threat in every piece of code. In this paper, we present various facets of buffer overflows and pinpoint their practical relevance. Despite numerous protection mechanisms it remains difficult to protect against buffer overflows in their entirety.

This paper gives an overview of the existing buffer overflow techniques, emphasizing the attacker's perspective. We finish with a discussion of the most well-known mitigation techniques.

Contents

1 Introduction

Software vulnerabilities can emerge from a wide range of possibilities. According to the CERT Coordination Center [Cer], the most prevalent origin remains *buffer overflows*. A buffer overflow occurs when a program attempts to store data in a buffer and the data is larger than the size of the buffer [Szo05]. Writing beyond the buffer boundaries results in an undesired modification of adjacent memory locations. This data corruption can be exploited by an attacker to change the control flow of the program.

The vast majority of malicious software (*malware*) leverages buffer overflows to execute subsequent attacks. After an initial infection, a malicious program may download additional code pieces to enhance its attack arsenal or update existing code [HM04, Szo05]. A recent example are *botnets*, a network composed of infected *zombie* hosts. Each instance can enhance its functionality by fetching additional attack modules [BY06].

Given that buffer overflows lay the foundation stone for any further exploitation, it is imperative to understand their nature and how to mitigate their impact.

Besides buffer overflows, several other types of vulnerabilities play an important role. For example, *format string vulnerabilities* form a separate class of software flaws. Due to misuse of functions for formatted output, such as the ANSI C function `printf()`, it is possible to modify the control flow of the program and point it to previously injected code [Scu01]. If an application is critically dependent on the temporal sequence of events, it might be susceptible to *race conditions*. Particularly filesystem races pose a serious threat for applications running in privileged context [BJSW05]. Handling integers proves difficult as well: when integers are used as memory offsets or involved in perform pointer arithmetic, their corruption can lead to control flow manipulation [Ble02]. There are many more classes of software vulnerabilities but an in-depth discussion of these goes beyond the scope of this paper.

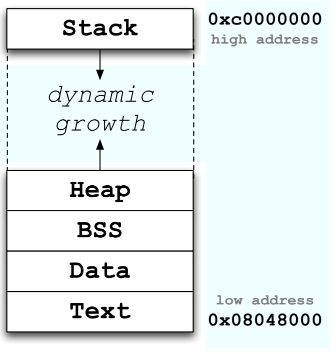
Because sophistication and diversity of buffer overflows continuously advances, we focus on fundamental approaches, rather than on cutting-edge exploitation techniques. Note that the presented techniques are not at all exhaustive, they only comprise a well-explored subset.

The remainder is organized as follows. We provide fundamentals and basic knowledge in §2. Section §3 discusses the evolution of buffer overflows and introduces the different concepts and exploitation techniques of each vulnerability. Having disputed different ways of attacking, we turn in §4 to countermeasures and protection mechanisms. We finish with a summary and conclusion in §5.

2 Basics

Before we discuss various buffer overflows and their exploitation techniques, we need to familiarize with some general concepts. It is difficult to understand the nature of buffer overflows without the understanding of function calls and operating system basics. In this section, we touch the very fundamental concepts required to understand the remainder. Therefore, we introduce the memory organization of processes in §2.1, recapitulate

Figure 1 Process layout in memory.



function calls in §2.2, and become acquainted with related terminology in §2.3.

2.1 Process Memory Organization

In order to understand the different types of buffer overflows, it is imperative to understand how the operating system handles processes in memory. There exist many formats for binary files. While Microsoft platforms use the *Portable Executable (PE)* format for binary files, all major UNIX flavors use the *Executable and Linking Format (ELF)* [Lev99]. If a program is launched, the binary code is loaded into memory and then executed. Such an executing program including the current values of the program counter, registers, and variables, is called a *process* [TW06]. Modern operating systems provide a separate virtual address space for each process. The *Memory Management Unit (MMU)* maps virtual address to physical addresses when necessary.

An executing process is partitioned into several *segments*, as shown in Figure 1. We briefly sketch the most important segments below.

Text segment. The *text segment* contains the binary instructions that are executed by the CPU. To prevent an uncontrolled modification of the instructions, this segment is read-only. An attempt to write to this segment would result in a segmentation fault.

Data segment. Global initialized and static data resides in the *data segment*. For example, the static global assignment `static int foo = 0` resides in the data segment, whereas `static int bar` lays in the BSS segment (see below).

BSS segment. Global uninitialized variables are stored in the *Block Started by Symbol (BSS) segment*. The operating system generally assigns such variables a value of 0 before handing control over to the process. This ensures that variables do not contain unexpected values. Although variables have a value after the assignment, they are still treated as uninitialized.

Heap. A program may resort to dynamic memory during runtime. This type of data is located on the *heap*. As an example, dynamic memory management in C is performed with the functions `malloc()` and `free()`.

Stack. Apart from global variables, a program contains features local variables inside functions. The default storage class of local variables is *automatic* (dynamic), i.e. they cease to exist after the function returns. On the other hand, static local variables reside in the data or BSS segment. The *stack* is the place where dynamic local variables are put together with function parameters and process control information.

It is important to note the stack starts at *high* addresses, and grows dynamically towards the heap (which in turn grows in the opposite direction). As we will see later, buffer overflows can occur in every writable segment.

2.2 Function Calls

High-level languages like C feature independent code sections, so called *functions* (or *procedures*) to enable structured programming. A function performs a particular task and returns thereafter to its original position. The information required by a single function execution is stored in an contiguous block of data called *activation record* or *frame* [ASU86]. Languages like Pascal and C push the activation record on the stack when entering a function, and pop the activation record off the stack when returning control to the caller.

An activation record includes the entire function context: function parameters, saved machine status, and local data. With respect to buffer overflows, the saved machine status is particularly sensitive since it contains information which have an impact on the program's control flow. Attackers commonly seek to manipulate these to gain control of the program.

Throughout this paper, we exclusively discuss the IA-32/x86 architecture because it is the most widespread and well-explored platform. The activation record on this architecture begins with the *return instruction pointer (RIP)* and the *saved frame pointer (SFP)*[1]. The RIP is the address of the next instruction in the code segment after the function returns. On the x86 architecture, the instruction pointer resides in the register `%eip` (*extended instruction pointer*). The SFP represents the address of the beginning of the previous stack frame, stored in the `%ebp` register (*extended base pointer*).

We now walk through an exemplary function call, as listed in Figure 2. The C source code is shown in Figure 2(a) and the corresponding assembler code shown in Figure 2(b). Let us follow the execution at the beginning of `main()`. Every function begins with a *prologue* which pushes the current frame pointer and sets the new frame pointer to the current stack pointer (located in the `%esp` register on the x86 architecture). This step, shown in Figure 3(a), saves the current stack environment. *OFP* denotes the *old frame*

[1]Unlike Aho et al. [ASU86], we chose to begin the activation record with the RIP and SFP because this view is more similar to the actual stack layout.

Figure 2 A simple function call example.

```
void foo(int a, int b, int c)
{
    int bar[2];
    char qux[3];

    bar[0] = 'A';
    qux[0] = 0x2a;
}

int main(void)
{
    int i = 1;
    foo(1, 2, 3);

    return 0;
}
```

(a) C source.

```
main:                   foo:
  pushl %ebp              pushl %ebp
  movl  %esp,%ebp         movl  %esp,%ebp
  subl  $4,%esp           subl  $12,%esp
  movl  $1,-4(%ebp)       movl  $65,-8(%ebp)
  pushl $3                movb  $66,-12(%ebp)
  pushl $2                leave
  pushl $1                ret
  call  foo
  addl  $12,%esp
  xorl  %eax,%eax
  leave
  ret
```

(b) Assembler source.

pointer from the previous stack frame, which is equal to the value of the SFP. The prologue further extends to the memory reservation for local variables (`subl $4,%esp`, because the local variable `i` is a 4-byte integer). After assigning 1 to `i`, the parameters for the function `foo()` are pushed on stack in reverse order. Function parameters are always referenced with relative offsets to the frame pointer[2]. Right before branching into `foo()` the RIP is pushed on the stack (Figure 3(b)).

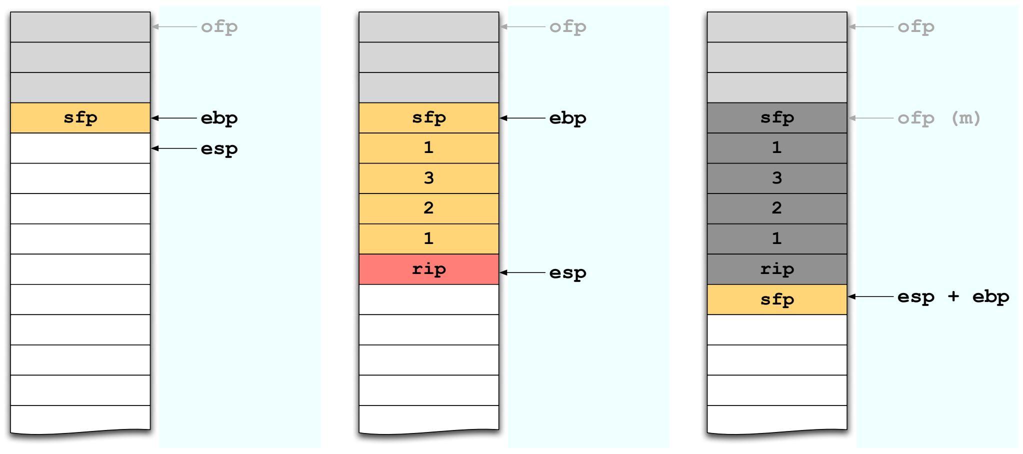
The function `foo()` begins with the prologue as well (Figure 3(c)). Note that the stack operates on 32 bit *dwords* and thus rounds up the required space for local variables. Hence, 12 bytes instead of 11 are allocated. Having assigned the values to the local variables (Figure 3(d)), the *epilogue* of the function, comprising the instructions `leave` and `ret`, is executed. The `leave` instruction realigns the stack pointer to the frame pointer, thereby restoring the old stack frame. Afterwards, the stack looks like in Figure 3(e). The `ret` instruction is equivalent to `popl %eip`, popping off the RIP into the program counter and continuing in the previous stack frame (Figure 3(f)).

Being back in the context of `main()`, it is time to clean up the stack. Since the function parameters are pushed by the caller, they have to removed by the caller as well. Therefore, 12 bytes are added to the value of `%esp` (three 4-byte parameter). Finally, the program writes the return value 0 into the general-purpose register `%eax` and follows the same leaving scheme as discussed in function `foo()`.

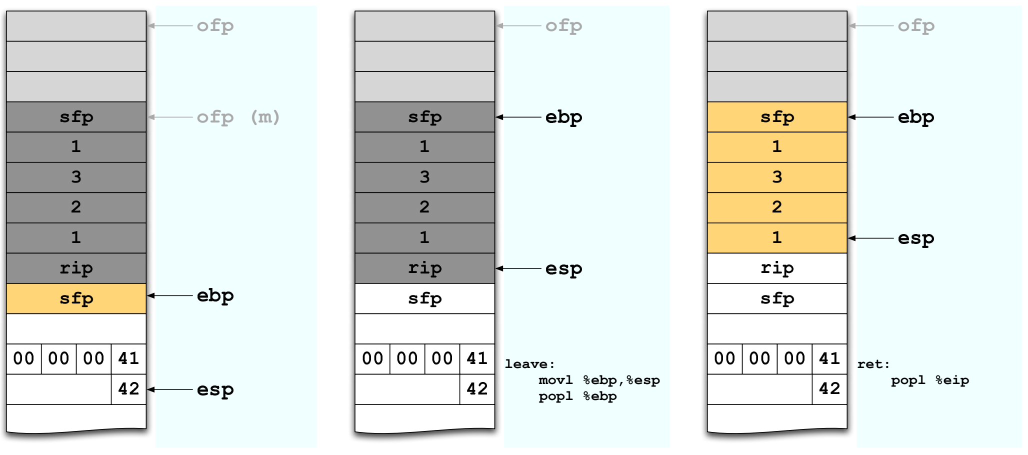
Understanding function calls is fundamental when dealing with buffer overflows. In §3.1, we discuss exploitation techniques which make extensive use of activation record manipulation.

[2]However, compiler optimizations such as `-fomit-frame-pointer` calculate absolute address values for performance issues, rendering the frame pointer in most cases superfluous.

Figure 3 Stack layout for Figure 2.



(a) `main()` prologue. (b) Calling `foo()`. (c) `foo()` prologue.



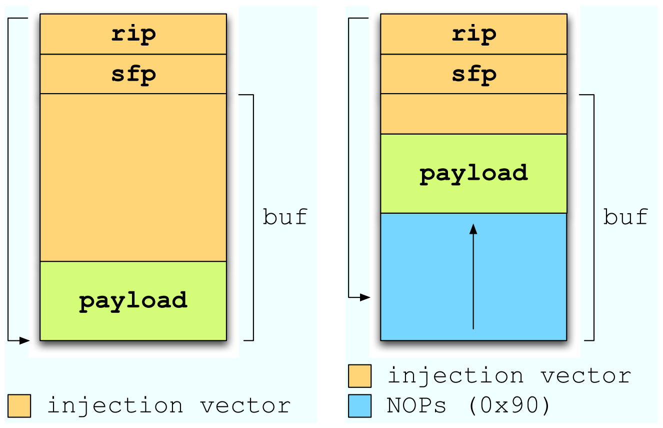
(d) . (e) . (f) .

2.3 Buffer Overflows and Exploits

Since many years, *buffer overflows* (or *buffer overruns*) are the most frequent exploited software vulnerability. When a program attempts to write data into a buffer that is larger than the buffer size, a buffer overflow occurs. Runtime environments such as Pascal, Ada, Java, and C# can detect buffer overflows and generate exceptions. But runtime environments geared towards performance, like C and C++, do not perform such checking. Because most system software is written these languages, numerous critical applications are error-prone and susceptible to buffer overflows.

Writing beyond the buffer boundaries alters adjacent data and can lead to undesired effects. If the next memory area contains important control information, such as the return address or a function pointer, an attacker can use the buffer overflow to alter these.

Figure 4 Exploit Anatomy.



(a) Classical Exploit.   (b) NOP Sliding.

Dependent on the supplied data, different types of attacks are possible. Overwriting the sensitive data with junk data refers to as a *Denial-of-Service (DoS)* attack, rendering the program unusable, or worse, causing the program to terminate. Equally dangerous is to overwrite the sensitive data systematically to change the control flow of the program. Attackers use this method to divert the execution path to their own malicious code. If the injected code contains instructions that spawn a shell, it is referred to as *shellcode*.

A piece of code that makes use of an existing vulnerability is termed an *exploit*. If exploit code is published the same day the vulnerability is disclosed, we speak of *0-day* ("zero day") exploits. To defend against such attacks is extremely difficult because vendors usually do not provide patches betimes.

Classical exploits typically consist of two parts: the *injection vector (IV)* and the *payload*, as shown in Figure 4(a). The injection vector diverts the control flow to the payload which contains the malicious instructions. The attacker's goal is to execute the payload, whereas the IV paves only the way for it. One can think of the IV as "a cruise missile for the warhead (payload)" [Kle04]. Hoglund and McGraw describe the IV as

> (1) a structural anomaly or weakness that allows code to be transferred from one domain to another, (2) a data structure or medium that contains and transfers code from on domain to another" [HM04].

This modularization enables flexible exploit creation: once the desired payload is crafted, it can be combined with different IVs. In other words, the set of all IVs describes the existing exploitation techniques.

A common enhancement to increase the probability of a successful exploitation is *NOP sliding*, depicted by Figure 4(b). In most cases, the absolute address of the buffer is not known[3]. It is thus difficult to jump to the very beginning of the payload. But to

[3]Stack addresses are in general difficult to guess, e.g. due to varying environment variables or address

successfully execute the injected code, one has to start with its first byte. Therefore, a series of NOP instructions is prepended to the payload, only incrementing the instruction pointer withouth affecting anything else. If the control flow is now diverted anywhere into the NOP field, the processor "slides" down to the beginning of the payload.

It is further possible to design multiple platform NOP slides that support different processor architectures [HM04]. For example, an x86 and MIPS compatible NOP slide includes the code bytes `24 0F 90 90`, because the MIPS features 32-bit instructions. This sequence translates to `0x9090` on a MIPS, but to an innocuous `add` on an Intel architecture.

The interested reader may consoult the literature for many more sophisticated techniques to effectively reach the payload of an exploit [HM04, Kle04, Szo05].

3 Types of Buffer Overflows

This section provides an overview of several exploitation techniques that evolved over time. In literature, the described techniques are sometimes categorized into *generations* of buffer overflows [Kle04, Fla02, Szo05], because the discovery of the first lead to the discovery of the next. Other authors prefer to categorize the techniques based on data structures and their associated algorithms [LC03, CWP$^+$00]. We adopt the latter categorization.

The exploitation of buffer overflows is often bound to a specific execution environment. We thus have to delve into particularities of a selected architecture. Note that other architectures might exhibit equal weaknesses that can be exploited in a similar manner.

In the following, we discuss two general types of buffer overflows, *activation record hijacking* in §3.1 and *pointer subterfuge* attacks in §3.2.

3.1 Activation Record Hijacking

The first type of buffer overflows tamper with the *activation record* [ASU86] on the stack. Each time a function is called, management information are pushed on the stack, such as RIP and SFP (see §2.2). There exist various techniques based on activation record manipulation, yet they all originate from the well-known "stack smashing" attack. Although buffer overflows are not a stack problem per se, the stack provides easy access to the instruction pointer via the RIP.

We first present "stack smashing" in §3.1.1, followed by more advanced *frame pointer overwrites* in §3.1.2. Thereafter, we look at *arc injection* attacks in §3.1.3.

3.1.1 Stack Smashing

The first generation of buffer overflows, named "stack smashing", was introduced in 1996 by Aleph One [One96]. As the name suggests, the vulnerability involves tampering with the stack environment of the process. Recall that the RIP is saved on the stack

space randomization (see §4.2).

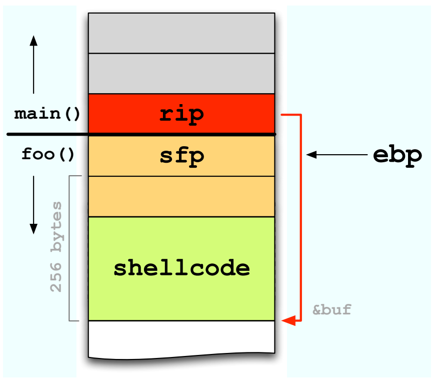
Figure 5 A stack-based buffer overflow.

```
void foo(char *args)
{
    char buf[256];
    strcpy(buf, args);
}

int main(int argc, char *argv[])
{
    if (argc > 1)
        foo(argv[1]);

    return 0;
}
```

(a) Vulnerable code.



(b) Exploiting the vulnerability.

before branching into a function (see §2.2). If a local fixed-size buffer (e.g. an array) is used in an unsafe manner, overflowing the buffer allows an attacker to overwrite the RIP with an arbitrary value. When the function returns, the modified RIP is loaded in the instruction pointer register, resulting in a change of the control flow. Clearly, the goal of the attacker is to divert the control flow to a custom location, e.g. to execute previously inserted instructions.

The following example, depicted by Figure 5, illustrates this vulnerability. In the code shown in Figure 5(a), the function `foo` copies a passed string into a local buffer `buf`, using the unsafe `strcpy()` function which continues to copy characters until it reads a `\0`. Given that the function `foo()` is invoked with arbitrarily arguments from the command line, an attacker can overflow the the buffer `buf` and thus modify the adjacent memory space.

In this example, the local buffer is declared at the beginning of the function. Consequently, the memory directly after the buffer holds the SFP and the RIP (see §2.2) which can now be modified by the attacker. Figure 5(b) shows a possible exploit scenario where the RIP is overwritten with the address of `buf`. When the function returns, the corrupted RIP is popped into the instruction pointer register and the attacker-supplied shellcode is executed[4].

3.1.2 Off-by-Ones and Frame-Pointer Overwrites

The second generation of buffer overflows is known as *off-by-ones* and *frame-pointer overwrites*. An off-by-one error occurs when starting at 0 instead of at 1, or by comparing $<= n$ instead of $< n$ (or vice versa). More specifically, an off-by-one *overflow* specifies a one-byte buffer overflow. Such an error is made exceedingly often in loop conditions:

[4]For the sake of simplicity, we chose a naive exploitation technique where the RIP is overwritten with the address of the buffer.

```
void bar(char *data)
{
    char buf[256];
    int i;

    for (i = 0; i <= 256; i++)
        buf[i] = data[i];
}
```

This inaccurate loop provides access to the element `buf[256]` which is already beyond the buffer boundary, leaving one byte at the mercy of the attacker. Such a special type of an off-by-one overflow is termed a *frame pointer overwrite* because it is possible to modify one byte of the SFP[5].

In the following, we detail how an attacker can leverage this vulnerability to modify the program's control flow and execute own code [klo99]. The fundamental difference to "stack smashing" is that the RIP remains untouched. Instead, the SFP is manipulated in such a way that it entails a corruption of the higher stack frame, which in turn can be exploited to divert the control flow of the program.

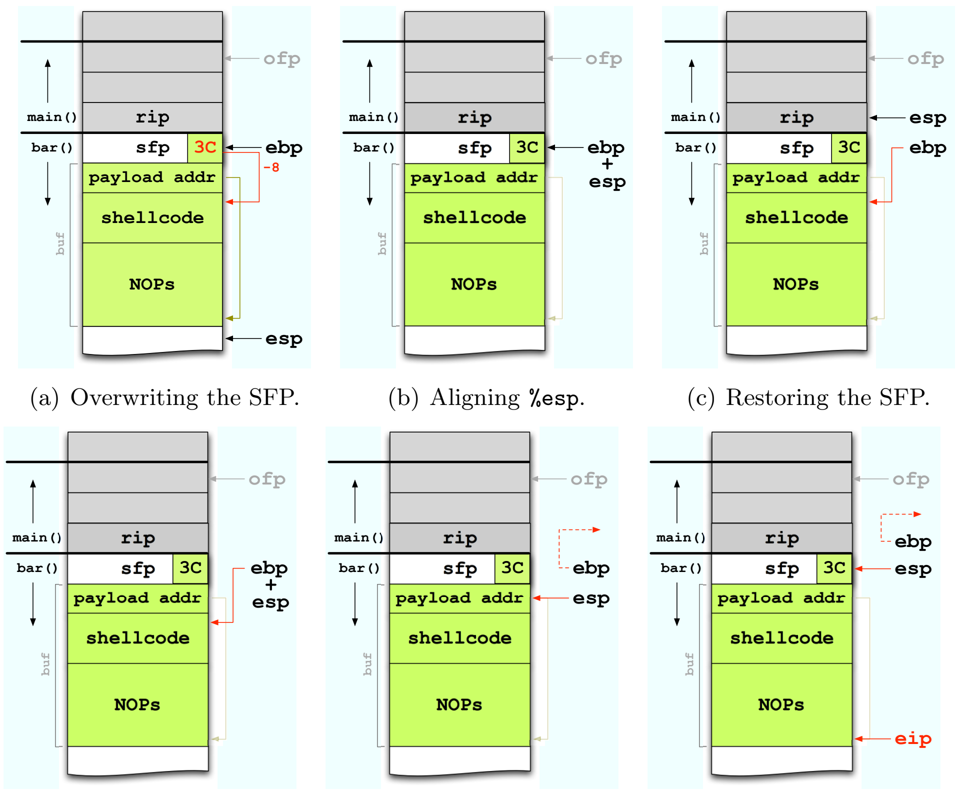
We start with a stack snapshot of the overflowed buffer in function `bar()` from the above code example. Figure 6(a) displays the exploit buffer and the overflowed LSB of the SFP whose address points to the cell 8 bytes below the SFP. The next instruction is `leave` in function `bar()`, which is equivalent to `mov %ebp %esp` (Figure 6(b)) and `popl %ebp` (Figure 6(c)). When returning to main, the RIP remains untouched, but the stack frame has been corrupted. Instead of pointing to the old frame pointer, `%ebp` points now to the previously altered SFP. The same cycle repeats when leaving `main()`. At first, `%esp` is realigned to `%ebp`, as shown in Figure 6(d). Both `%ebp` and `%esp` now point to the last 4 bytes of the shellcode. From the processor's perspective, the data at this location contains invalid instructions: `%ebp` points to an invalid address (Figure 6(e)). However, at this stage of exploitation, an invalid frame pointer does not matter any more. The next instruction, `ret`, loads the payload address into `%eip`, resulting in the desired change of the control flow. From now on, the program is doomed to execute the injected instructions, beginning with a NOP slide and finishing with the shellcode.

3.1.3 Arc Injection

Arc Injection [PB04, Sea05] is a more general term for *return-into-libc* [Des97, Ner01] exploitation techniques. Instead of inserting executable instructions into a vulnerable buffer, it is also possible to supply data that lead to the desired effect when the program operates on it. The term "arc injection" refers to the exploitation approach. In contrast to injecting executable instructions, arc injection techniques only insert a new arc (control flow transfer) into the control flow graph, as opposed to inserting also a new node.

[5] To be more precise, on architectures with *little-endian* byte order, such as x86, the *least significant byte (LSB)* of the frame pointer is being overwritten.

Figure 6 The frame pointer overwrite.



(a) Overwriting the SFP. (b) Aligning `%esp`. (c) Restoring the SFP.

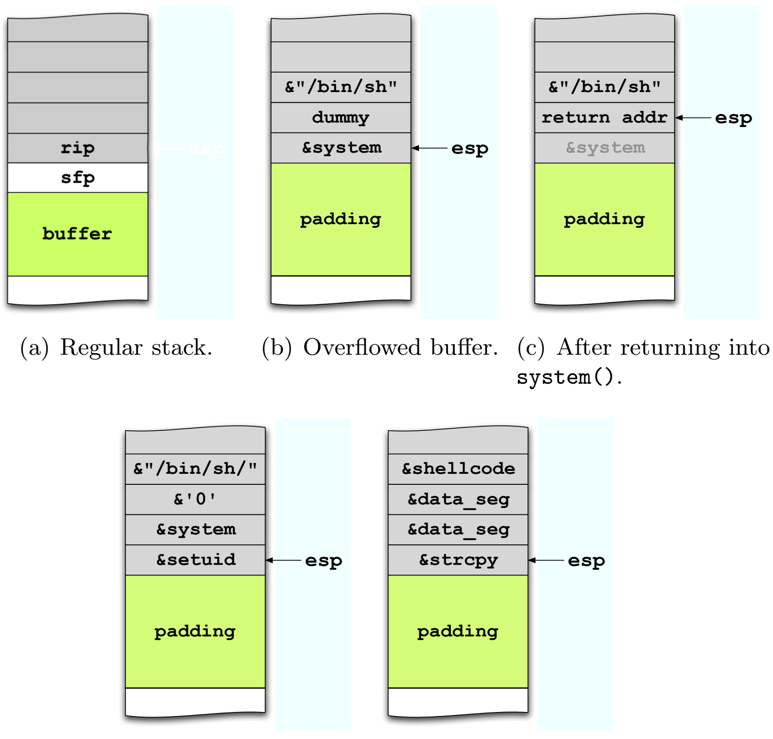
(d) Aligning `%esp`. (e) Ready for `popl %eip`. (f) Sliding to hell.

A common example involves manipulating a string representing a command line. If this string is used by the program to spawn another process, it is possible to execute arbitrary code on the system under attack. To this end, the RIP is also altered, but the control flow is diverted to a C library function rather than to shellcode. This type of attack is visualized in Figure 7.

First, the RIP is overwritten with the address of `system()`. Moreover, the address of its argument `"/bin/sh"` is part of the attack code (Figure 7(b)). Clearly, the absolute addresses of `system()` and the string `"/bin/sh"` need to be known in advance. Since the C library is in most cases linked dynamically, gathering the address of `system()` involves finding out the position where the C library is mapped in address space[6]. On the other hand, supplying the string `"/bin/sh"` is straight-forward: it merely involves creating an environment variable and referencing its address. After having overflowed the vulnerable buffer, `system()` takes the attacker-provided string as argument and spawns a shell. Note that the `system()` function requires a return address to jump to after it

[6]To this end, one can employ debuggers or have a look at the */proc* directory. However, mapping shared libraries at random addresses defeats this method [PaX].

Figure 7 Arc injection example.



(a) Regular stack. (b) Overflowed buffer. (c) After returning into `system()`.

(d) Function chaining. (e) Shellcode in a writable segment.

has finished its job. But since the program will never reach that point, we can safely chose an arbitrary dummy value (Figure 7(c)).

A more sophisticated use of arc injection includes chaining the invocation of multiple functions. Rather than just providing a dummy value, supplying a valid return address for the C library function allows calling functions in a row. For example, overwriting the RIP with the address of `setuid()`, immediately followed by the address of `system()` allows an attacker to first escalate his privileges via `setuid()` and then spawn a shell via `system()` (shown in Figure 7(d)).

Further, an attacker can leverage function chaining in order to place shellcode in the data segment and transfer control to it [Woj98]. Figure 7(e) illustrates this example. Here, the RIP is overwritten with the address of `strcpy()`, together with the address of the shellcode in the data segment.

However, function chaining is limited due to the complexity in placing their return addresses and function parameters. Consider our example in Figure 7(d) where `system()` crashes when it returns, because its return address serves as a parameter for `setuid()`. Nonetheless, there exist methods like *stack pointer lifting* and *frame faking* [Ner01] to circumvent this limitation.

Arc injection exploits prove especially useful in environments with *non-executable* stack environments (see §4.2). Since no attacker-provided code is executed, these protection mechanism fall short in detecting such attacks.

3.2 Pointer Subterfuge

The deliberate modification of the value of a pointer is referred to as *pointer subterfuge* [PB04]. As these types of attacks modify directly the control flow of the program, they are also known as *control flow attacks*. Originally, pointer subterfuge attacks were developed to evade stack protection mechanisms. Nowadays, they evolved to an equally dangerous class of attacks showing up in manifold variations. We now sketch some well-known types of pointer subterfuge.

Function-pointer clobbering. Overwriting *function pointers* allows an attacker to jump to its own code, as shown in Figure 8(a). In function `foo`, a function pointer `(*f)` is declared directly after the fixed-size buffer `buf`. Since `memcpy` does not perform any boundary checking, it is possible to write beyond `buf`'s boundary. An attacker could overwrite `f` with the address of `buf`, resulting in a control transfer to `buf` on the next call of `f`.

Another example is the *Structured Exception Handling (SEH)* mechanism from Microsoft. A raised exception entails the invocation of a corresponding handler, whose function pointers are stored in a linked list of handlers on the stack. Hence a buffer overflow on the stack allows to take advantage of this vulnerability.

Although the above example utilizes a stack-based buffer overflow, it can also occur in other writable segments, such as BSS or heap. Function pointer clobbering is thus a very useful technique when the program employs stack protection mechanisms (see §4.2).

Data-pointer modification. A more generic concept is data-pointer modification [PB04]. If an attacker controls an assignment in which the destination is an lvalue, he can perform an *arbitrary memory write*. Figure 8(b) depicts such a scenario, where both the value `val` and the corresponding target `ptr` of a later assignment is declared directly after the buffer `buf`. By overflowing the buffer, the attacker controls the lvalue and rvalue of the assignment, giving him the opportunity to perform an arbitrary 4-byte memory write[7]. Many protection mechanisms can be effectively circumvented with this powerful technique.

Data-pointer modification combines very well with function-pointer clobbering [PB04]. As illustrated in Figure 8(b), the external function-pointer `(*f)()` is not a local variable. Therefore, it cannot be altered in the context of the buffer overflow. Assuming the attacker knows the address of `f`, he can leverage the data-pointer modification to change the value of `f`, yielding the same result as a function-pointer overwrite.

[7]The memory layout and data types may vary on different architectures.

Figure 8 Pointer subterfuge attacks.

```
void foo(void *arg, size_t len)
{
    char buf[256];
    void (*f)() = ...;

    memcpy(buf, arg, len);
    f();

    return;
}
```

(a) Function-pointer clobbering.

```
void bar(void *arg, size_t len)
{
    char buf[256];
    long val = ...;
    long *ptr = ...;
    extern void (*f)();

    memcpy(buf, arg, len);
    *ptr = val;
    f();

    return;
}
```

(b) Arbitrary memory write.

VPTR smashing. C++ features *virtual functions* to enable dynamic function dispatching. The majority of C++ compilers uses a *virtual function table (VTBL)* associated with each class to implement virtual functions. Every instantiated object has a *virtual pointer (VPTR)* to its corresponding VTBL. If an attacker overwrites an object's VPTR with a pointer to a forged VTBL, the next virtual function call leads to a control flow transfer [Rix00].

For example, the forged VTBL can include pointers to previously injected shell-code. If the VTBL has n entries, the attacker would overwrites each entry with the same address to maximize the chance of successful exploitation. A call to a virtual function translates to **(VPTR + x)* whereas x represents the number of the corresponding VTBL entry.

Although VPTR smashing has not yet been used often in the wild, it qualifies as an effective option to escape heap protection mechanisms [PB04].

As outlined above, all pointer subterfuge attacks prove very useful to complement other attacks. A skilled attacker might use several indirections and combine various techniques to circumvent memory protection mechanisms. Therefore, a thorough understanding of the attacker's arsenal is required to devise effective mitigation strategies.

Pointer subterfuge attacks can also occur in the BSS segment. Klein [Kle04] terms *BSS overflows* as the third generation exploits. However, the main-stream literature deems *heap overflows* (see below) as third generation. We thus touch the topic only briefly. Recall that uninitialized global and static data lays in the BSS segment (see §2.1). Unlike the stack, the BSS segment grows towards increasing addresses. With respect to buffer overflows, this implies that data *after* the vulnerable buffer can be manipulated. Conover presents an example where a file pointer is declared after a vulnerable buffer [Con99]. The pointer is then used to write to a temporary file. In a concrete exploitation scenario, an attacker can overwrite the pointer value and let it point to `/etc/passwd` in order to add a new privileged user to the system.

3.2.1 Heap Overflows

In the following, we present *heap overflows* and their consequences. Since the operations that lead to the vulnerability are pointer overwrites, we chose to discuss this topic in the context of pointer subterfuge attacks.

The heap is a coherent memory area available for allocation and deallocation of arbitrary-sized blocks (see §2.1). Like the stack, the heap has to manage internal status information along with the actual data in the same area. Deliberately modifying this management information using a buffer overflow refers to as a *heap overflow*.

As heap overflows inherently rely on the implementation of the dynamic memory allocator, we first introduce the internals of the `malloc()` implementation of C library. Since many years, the C library employs *ptmalloc* as standard implementation for dynamic memory allocation. It partitions the heap in reserved memory blocks of arbitrary size, called *chunks*. Chunks can be allocated, freed, split, and merged with other chunks. To this end, the C library features functions such as `malloc()`, `calloc()`, `realloc()`, `free()` and friends. Contrasting to allocated chunks, free chunks can never lay next to each other. Instead, they are merged together into one bigger free chunk to reduce the overall number of small unusable chunks.

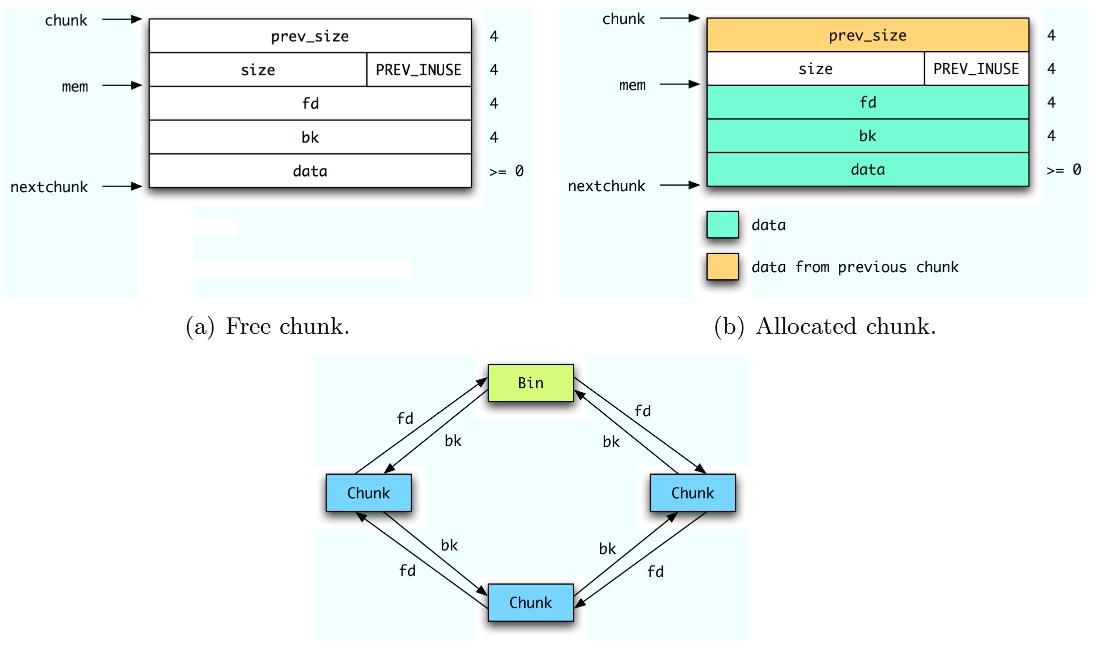
In addition to the actual chunk data, meta information about the exact size and position of the chunk has to be managed. On the one hand, this information is used to merge empty chunks right next to each other. On the other, it is used to locate a particular chunk when starting from an arbitrary chunk. These meta information are called *boundary tags*, because they embrace a chunk with data after and in front of it.

Allocated and free chunks have different boundary tag structures, as shown in Figure 9. First, we look at commonalities that both types share. Each boundary tag has associated three pointers: *(i)* the *chunk* pointer, marking the beginning of a chunk including the entire boundary tag, *(ii)* the *mem* pointer, returned to the program requesting memory via `malloc()`, and *(iii)* the *nextchunk* pointer, specifying the beginning of the next chunk.

A free chunk, depicted by Figure 9(a), uses the *prev_size* field to indicate the size (in bytes) of the previous chunk if it is free. Otherwise it is used by the previous chunk to to store data, trying to use the available memory as efficiently as possible. The *size* field contains the absolute size of a chunk, from *chunk* up to *nextchunk*. Further, the least significant bit, *PREV_INUSE*, indicates if the previous chunk is allocated or not. The next two fields, *fd* and *bk*, are only used by free chunks[8]. They stand for the *forward pointer* and *back pointer*, connecting a free chunk with the previous and next free chunk. Since free chunks do not lay next to each other, these pointers do not represent physical vicinity but rather a special arrangement in *bins*. Bins are essentially doubly-circularly-linked lists. A schematic view of a bin with three chunks is shown in Figure 9(c). In general, bins contain free chunks of similar size, but bins for sizes $<$ 512 bytes contain chunks of all the same size, spaced 8 bytes apart. Larger bins are approximately logarithmically spaced.

[8]An allocated chunk, as illustrated in Figure 9(b), uses the *fd* and *bk* fields for data as well.

Figure 9 Different chunk structures.



(a) Free chunk.

(b) Allocated chunk.

(c) A bin with containing chunks.

The exploitable vulnerability occurs when a chunk is removed from bin, e.g. to merge it with adjacent free chunk. Removing a chunk is done with `unlink()` macro, which is defined as follows:

```
#define unlink(P, BK, FD)
{
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
}
```

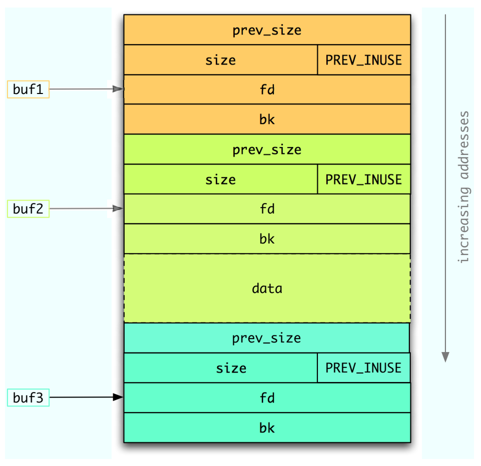
In this piece of code, `P` denotes the chunk, `BK` the back pointer, and `FD` the forward pointer from the chunk being unlinked. The operations themselves are self-explanatory. We note though, that `FD->bk` is equivalent to `FD + 12` and `BK->fd` is the same as `BK + 8` (see Figure 9). This will later play an important role when tampering with pointer values.

We now turn to the actual vulnerability. Therefore, consider the example code in Figure 10(a): Three memory allocations followed by a copy operation, and finally the obtained memory is freed. Remember that the three buffers are separated by their boundary tags; their memory layout is shown in Figure 10(b). Similar to activation

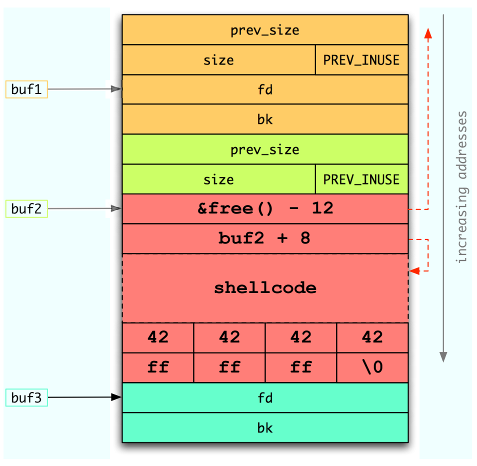
Figure 10 Exploiting `unlink()`.

```
...
char *buf1 = malloc(0);
char *buf2 = malloc(256);
char *buf3 = malloc(0);
...
gets(buf2);
...
free(buf1);
free(buf2);
free(buf3);
...
```

(a) `unlink()` vulnerability.



(b) Before `gets(buf2)`.



(c) After `gets(buf2)`.

record hijacking, we can overwrite management information on the heap. The idea is to manipulate the *fd/bk* fields of `buf2` and then call `unlink()` on the modified chunk by flipping the *PREV_INUSE* bit of `buf3`.

When `free(buf1)` is called, it looks at the next chunk to see whether it is in use or not. If the next chunk is unused, `unlink()` is called to merge it with the chunk being freed. To this end, the *PREV_INUSE* bit of the third chunk is evaluated.

Clearly, calling `unlink()` on a allocated chunk is not intended and causes most likely a segmentation fault. In our example, the attacker has control over the relevant data. As illustrated above, `unlink()` dereferences `(fd + 12)` in order to write at that address the value `bk` (alleged value of the back pointer). As a result, an arbitrary 4-byte memory write is possible.

What are potential candidates for such an overwrite? Certainly, a pointer is chosen to modify the control flow of the program. The RIP on the stack might look attractive, but it is difficult to map out. Another candidate is a function pointer in the *global offset table (GOT)*. The GOT is created by the linker and contains pointers to all global data that an executable addresses [Lev99]. Disassembling the binary (e.g. with `objdump`) yields the desired function address in the GOT.

Let us examine the exploit buffer in Figure 10(c). The *fd* field is overwritten with `&free() - 12`, meaning the address of `free()` in the GOT minus 12. Since `unlink()` adds an offset of 12 to reach the *bk* field of the next chunk (`P->fd->bk`), the offset has to be subtracted. The right side of the assignment is the *bk* field, which is set to `buf2 + 8`, because the shellcode is located 8 bytes away from `buf2`. Beyond the actual buffer boundary, 4 dummy values `0x42` overwrite the *prev_size* field. At last, the terminating `\0` flips the *PREV_INUSE* bit, setting the ball rolling. With the next call of `free()`, the program executes the shellcode instead of freeing the desired memory.

Upon closer examination, we have so far ignored one `unlink()` directive where the back pointer is updated (`BK->fd = FD`). BK is clearly overwritten with `buf2 + 8`. Further, adding the `fd` offset entails a memory write at `(buf2 + 8) + 8`, which is in the middle of the shellcode. The shellcode has thus to jump over its modification in order to spawn a shell successfully.

4 Countermeasures

To err is human. Exploitable software flaws produced by software developers will not cease to exist. It is therefore indispensable to confront buffer overflow pro-actively.

There exist generally two approaches to counter buffer overflows [EL02]. The first class of mitigation techniques, presented in §4.1, involves eliminating the cause of buffer overflows. The second class, discussed in §4.2, deals with alleviating the impact of buffer overflows by fixing the surroundings of a vulnerable program, i.e. containing the resulting damage.

4.1 Eliminating Flaws

Techniques that detect and correct human errors before the software is deployed greatly reduce the potential risk of exploitation. Numerous approaches exist that focus on extincting bugs before they can be exploited. In the following, we briefly introduce some basic aspects presented in [Kle04].

Secure Programming. Especially languages that focus on performance and flexibility rather than security and reliability are prone to programming errors. Even worse, direct low-level memory access and unsafe library functions potentiate numerous vulnerabilities. It is eventually up to the programmer to produce safe code. For example, an educated programmer should never use unsafe library functions, such as `strcpy()`, that do not perform explicit boundary checks of fixed-size buffers.

Source Code Audit. Apart from secure programming paradigms, a comprehensive source code analysis can uncover many software flaws. Such a line-by-line audit allows selective checking and fixing of known pitfalls. However, human inspection is costly, error-prone, and time-consuming. Since many tasks can be performed, automatic software tests complement the testing process particularly for complex processes.

Automatic Software Tests. Enhancing time-consuming human inspection with automated software tests proves apt for more complex scenarios. The goal of automated tests is to accommodate the knowledge of experts in tools to scrutinize software in a faster and more cost-efficient fashion than manual inspection.

Source code analyzer fan out into *static* and *dynamic* approaches. While static analysis tools inspect on the source of the program, dynamic tools check for run-time errors. Further, static tools can be subdivided in *lexical* and *semantic* analyzers. The former treat source code as individual tokens, where each token is verified separately. Tools such as *grep*, *flawfinder* [Dav], and *RATS* [For07] belong to this class. The latter incorporate semantic context gathered from a control flow analysis to derive conclusions about connected activities. For example, compiler warnings spit out semantic errors, and tools like *splint* [LE01] come with a rich set of functions for contextual error analysis.

Dynamic approaches execute the program and try to find run-time errors leading to buffer overflows. The tools for this task are referred to as *tracer*. They log the control flow and analyze the audit trail. A debugger is the most prominent example for a tracer. Many tools exist that can be employed to conduct run-time analysis, e.g. *Electric Fence* [Bru], *Purify* [IBM], and *valgrind* [NS07].

Binary Audit. If the program being tested is only available as a binary, the security analysis turns out to be more difficult. Without the source code, one uses techniques such as *fault injection* or *reverse engineering*. Fault injection deliberately perturbs the environment to provoke obvious program flaws that suggest about internal errors. To this end, the program is fed with input generated by *fuzzers*, that create random data to find buffer overflows by chance.

Reverse engineering on the other hand involves examining the binary program itself, rather than only tampering with its environment. Disassemblers such as *IDA Pro* [Dat] and *SoftICE* [Cor99] visualize assembler instructions in an interactive GUI. The comfortable interface allows the analyst to reconstruct a detailed picture from the program's internals, rendering disassembly an extremely effective yet complex method to find software vulnerabilities.

4.2 Limiting Damage

Relying solely on flaw finder tools, human code reviews, or automatic software tests might iron out numerous bugs, but cannot provide complete safety. For example, hitherto unknown exploitation techniques cannot be addressed by these mechanisms. Therefore, a complementary approach to eliminating flaws is limiting the damage of a possible exploitation. For the following approaches, the program code itself is not the subject of analysis, instead one tries to cut all possible attack paths to the application.

Wrapping unsafe library functions. To mitigate the impact when using unsafe library functions, the *libsafe* [BST99] project pursues an approach that inserts a dynamically loadable library catching and substituting calls to vulnerable libraries. Based on the preload feature of dynamically loadable ELF libraries, libsafe transparently loads its own wrapper with processes it should protect.

Many types of stack-based buffer overflows can be eliminated using libsafe. However, it is by far not possible to include all potential unsafe library functions in this concept. Further, protecting against BSS and heap overflows fails because the libsafe only recognizes stack-based attacks (due to relative frame pointer referencing).

Compiler Extensions. Another approach enriches the compiler with specific safety mechanisms, ranging from RIP overwrite protections to comprehensive bounds checking.

StackGuard [CPM$^+$98], for example, features a stack protection mechanism that inserts a *canary* value between the SFP and RIP. On each function return, the canary's integrity is controlled. If the canary is manipulated, StackGuard assumes that the RIP was modified as well and terminates the program immediately. This mechanism only prevents RIP overwrites. We saw in §3.1.2 that the control flow can be easily modified without involving the RIP. *ProPolice* [Eto03], an enhanced StackGuard concept, extends the protection to the SFP and performs array variable relocation to the highest part of the stack frame. This makes it harder to overwrite them and corrupt other variables. Microsoft's pendant, based on a similar technique with a *security cookie*, is called */GS option* [Bra02].

All stack protections have one thing in common: they only protect the stack. Unfortunately, the stack is just one piece in the puzzle. As previously shown, pointer subterfuge attacks (see §3.2) can occur also in other segments. The limited scope of these compiler extensions shows that none of the approaches provide comprehensive defensive mechanism. In combination, these tools protect nonetheless against many types of buffer overflows and should not be dismissed just because no overall solution is provided.

Environmental Modifications. A completely different approach from the so far mentioned techniques deals with the modification of the program's execution environment.

One method transforms the stack into a *non-executable* segment. Most exploits are based on overwriting a function's RIP to point to some injected code which lays also on the stack. Setting the stack non-executable makes it much harder to exploit buffer overflows. Again, injecting code in a different segment easily evades this protection.

The *PaX* [PaX] project tries to offer a more comprehensive protection. For example, it introduces *PAGE_EXEC*, a kernel patch that enables non-executable pages, eliminating the possibility of executing code in pages which are supposed to hold data only. Further, since most exploit techniques rely on the knowledge of certain addresses in the attack program, PaX features *Address Space Layout Randomization (ASLR)* to force an attacker to guess the addresses. Included is randomization of the top of the task's kernel and userland stack, base address for `mmap()` requests that do not specify one, and the base address of the main executable.

Using PaX in combination with *segvguard* [Ope], a daemon that temporarily disables execution of certain programs to prevent bruteforcing, a very large share of exploitation techniques can be eliminated. On the downside, performance penalties up to 500% may result from *PAGE_EXEC*.

5 Conclusion

Buffer overflows still account for the largest share of software vulnerabilities. Particularly dangerous is the area of remote exploitable vulnerabilities, where attackers hijack hosts in the Internet to perform criminal activities on behalf of others.

In this paper, we set out to give a broad overview about existing techniques that exploit buffer overflows. Our discussion is dominated by the attacking perspective and touches defensive approaches only marginally, because a detailed coverage would go beyond the scope of this paper. Nonetheless, we introduce the most well-known mitigation techniques. Further, it is important to note that the techniques can be effectively used in tandem to perform more complicated attacks.

Over time, many exploitation techniques evolved, eluding even sophisticated protection mechanisms. Although approaches such as PaX raise the bar significantly, a complete and practical solution to defeat buffer overflows is yet to be found. Meanwhile, defending against this ubiquitous threat requires a combination of static and runtime solutions.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[BJSW05] Nikita Borisov, Rob Johnson, Naveen Sastry, and David Wagner. Fixing races for fun and profit: How to abuse atime. In *Fourteenth USENIX Security Symposium (USENIX Security 2005)*, August 2005.

[Ble02] Blexim. Basic Integer Overflows. *Phrack*, 60–10, December 2002. Avaliable from `http://www.phrack.com`.

[Bra02] B. Bray. Compiler security checks in depth. Technical report, Microsoft Corporation, 2002.

[Bru] Bruce Perens. Eletric Fence. `http://perens.com/FreeSoftware/ElectricFence/`.

[BST99] A. Baratloo, N. Singh, and T. Tsai. Libsafe: Protecting critical elements of stacks, 1999.

[BY06] Paul Barford and Vinod Yegneswaran. An Inside Look at Botnets. In *Proceedings of the Special Workshop on Malware Detection*, Advances in Information Security. Springer Verlag, 2006.

[Cer] CERT Security Advisories. `http://www.cert.org/advisories/`.

[Con99] Matt Conover. w00w00 on Heap Overflows, 1999. Avaliable from `http://www.w00w00.org/articles.html`.

[Cor99] Compuware Corporation. Debugging blue screens. Technical Paper, September 1999.

[CPM$^+$98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78. San Antonio, Texas, jan 1998.

[CWP$^+$00] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference & Exposition – Volume 2*, pages 119–129, January 2000.

[Dat] DataRescue. IDA Pro Disassembler. `http://www.datarescue.com/idabase/`.

[Dav] David A. Wheeler. FlawFinder. `http://www.dwheeler.com/flawfinder/`.

[Des97] Solar Designer. Getting around non-executable stack (and fix). Bugtraq mailing list, `http://www.securityfocus.com/archive/1/7480`, August 1997.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, /2002.

[Eto03] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks (ProPolice), 2003. `http://www.trl.ibm.com/projects/security/ssp/`.

[Fla02] Halvar Flake. Third Generation Exploitation, 2002. Avaliable from `www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt`.

[For07] Fortify Software. Rough Auditing Tool for Security (RATS). `http://www.fortifysoftware.com/security-resources/rats.jsp`, 2007.

[HM04] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code.* Addison Wesley, 2004. ISBN 0-201-78695-8.

[IBM] IBM. Rational Purify. `http://www-306.ibm.com/software/awdtools/purify/`.

[Kle04] Tobias Klein. *Buffer Overflows und Format-String-Schwachstellen.* dpunkt.verlag, 2004. ISBN 3-89864-192-9.

[klo99] klog. The frame pointer overwrite. *Phrack*, 55–8, November 1999. Avaliable from `http://www.phrack.com`.

[LC03] K. Lhee and S. Chapin. Buffer overflow and format string overflow vulnerabilities. *Software—Practice & Experience*, 33:423–460, 2003.

[LE01] David Larochelle and David Evans“. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10 th USENIX Security Symposium*, pages 177–190, August 2001.

[Lev99] John R. Levine. *Linkers and Loaders.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1558604960.

[Ner01] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 58–10, November 2001. Avaliable from `http://www.phrack.com`.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007).* San Diego, California, USA, June 2007.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49–14, November 1996. Avaliable from `http://www.phrack.com`.

[Ope] Openwall. segvguard. `ftp://ftp.pl.openwall.com/misc/segvguard/`.

[PaX] PaX. `http://pax.grsecurity.net/`.

[PB04] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004. ISSN 1540-7993.

[Rix00] Rix. Smashing C++ VPTRs. *Phrack*, 56–10, 2000. Avaliable from `http://www.phrack.com`.

[Scu01] Scut. Exploiting Format String Vulnerabilities, Version 1.2, September 2001.

[Sea05] Robert C. Seacord. *Secure Coding in C and C++ (SEI Series in Software Engineering).* Addison-Wesley Professional, 2005. ISBN 0321335724.

[Szo05] Peter Szor. *The Art of Computer Virus Research and Defense.* Addison-Wesley, 2005. ISBN 0321304543.

[TW06] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, 2006. ISBN 0131429388.

[Woj98] Rafal Wojtczuk. Defeating solar designer non-executable stack patch. Bugtraq mailing list, `http://www.securityfocus.com/archive/1/8470`, 1998.