Low-level Software Security: Attacks and Defenses

Úlfar Erlingsson

Microsoft Research, Silicon Valley
and
Reykjavík University, Iceland

November, 2007

MSR-TR-07-153

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Low-level Software Security: Attacks and Defenses

Úlfar Erlingsson

Microsoft Research, Silicon Valley
and
Reykjavík University, Iceland

Abstract. This tutorial paper considers the issues of low-level software security from a language-based perspective, with the help of concrete examples. Four examples of low-level software attacks are covered in full detail; these examples are representative of the major types of attacks on C and C++ software that is compiled into machine code. Six examples of practical defenses against those attacks are also covered in detail; these defenses are selected because of their effectiveness, wide applicability, and low enforcement overhead.

1 Introduction

Computers are often subject to external attacks that aim to control software behavior. Typically, such attacks arrive as data over a regular communication channel and, once resident in program memory, trigger pre-existing, low-level software vulnerabilities. By exploiting such flaws, these low-level attacks can subvert the execution of the software and gain control over its behavior.

The combined effects of these attacks make them one of the most pressing challenges in computer security. As a result, in recent years, many mechanisms have been proposed for defending against these attacks. However, these defenses, as well as the attacks, are strongly dependent on low-level minutiae, such as the exact semantics of high-level language constructs, the precise syntax of machine-code opcodes, and the layout of code and data into memory. Therefore, in the literature, it is rare to find the full details of concrete attacks, or precisely how particular defenses prevent those attacks. This tutorial paper aims to partially remedy this situation.

The remainder of this introductory section gives more background about low-level software security, as well as notes on the presentation of low-level details. Next, Section 2 gives four examples that represent some of the important classes of low-level software attacks. These attacks apply to software written in C and C++, or similar languages, and compiled into executable machine-code for commodity, x86 hardware. These attacks are described in enough detail to be understood even by readers without a background in software security, and without a natural inclination for crafting malicious attacks. Then, Section 3 explains in detail the motivation, detailed mechanisms, and limitations of six important, practical defenses. A final Section 4 offers a brief summary and discussion.

Throughout, the attacks and defenses are placed in perspective by showing how they are both facilitated by the gap between the semantics of the high-level language of the software under attack, and the low-level semantics of machine code and the hardware on which the software executes.

1.1 Low-level Software Security for Different High-level Languages

It is not only in languages such as C and C++ that the issues of low-level software security must be considered. Low-level attacks may be possible whenever software is translated from a higher-level language into a lower-level language, without a guarantee that this translation preserves the higher-level abstractions.

Software programmers express their intent using the abstractions of the higher-level language. If these abstractions are not preserved in low-level execution, then this discrepancy can cause the software to behave in unexpected ways. Often, an attacker will be able to exploit this discrepancy, and divert the low-level execution of the software to perform arbitrary functionality of the attackers choosing.

Unfortunately, in practice, compilers from high-level to low-level languages do not guarantee full abstraction. This is true even for type-safe languages, such as Java and Microsoft's C#. As a result, low-level software attacks are also possible in such languages.

For instance, in Java, variables can be declared to be "private" to classes, in such a way that only the code in the declaring class is able to access the variables. In the Java language, this is a strong guarantee that a programmer might rely on—e.g., to implement a bank account whose sum can be updated only through well-defined deposit and withdraw operations in the declaring class.

However, Java is executed indirectly through compilation into a lower-level language, "JVML bytecode", whose semantics are different, As a result, when a class declares nested classes its private variables are actually not private, but are accessible to all the code in a set of classes. Often, this set may be dynamically extended by the Java virtual machine that executes JVML bytecode; these new classes may originate from third parties, as long as the type safety of the JVML bytecode can be verified.

As a result, attackers may be able to directly craft low-level JVML bytecode that pretends to be in the set of nested classes, and thereby circumvents the high-level Java language semantics of the software [1]. In this case, private variables may be subject to arbitrary modification by code written by attackers—e.g., artificially inflating a bank-account sum that is stored in a private variable.

Similarly, C# code is compiled into "CLR bytecode" with different semantics that create possibilities for low-level attacks. For instance, it is impossible in C# to invoke object constructors more than once, so a variable initialized during construction, and not modified from then on, may be assumed to be immutable in C# software—and it is easy to see how programmers might depend on this property. However, at the lower level of CLR bytecode, an object constructor is just another function that may be invoked multiple times [28]. Thus, in fact, a "C# immutable" variable may be modified arbitrarily often by an attack that

```
int unsafe( char* a, char* b )
{
    char t[MAX_LEN];
    strcpy( t, a );
    strcat( t, b );
    return strcmp( t, "abc" );
}
```

(a) An unchecked C function.

```
int safer( char* a, char* b )
{
    char t[MAX_LEN] = { '\0' };
    strcpy_s( t, _countof(t), a );
    strcat_s( t, _countof(t), b );
    return strcmp( t, "abc" );
}
```

(b) A safer version of the function.

Fig. 1. Two C functions that both compare whether the concatenation of two input strings is the string "abc". The first, unchecked function contains a security vulnerability if the inputs are untrusted. The second function is not vulnerable in this manner, since it uses new C library functions that perform validity checks against the lengths of buffers. Modern compilers will warn about the use of older, less safe library functions, and strongly suggest the use of their newer variants.

operates at the lower level of CLR bytecode. Here, as in the case of Java, the semantic gap between languages allows attackers to introduce valid, type-safe low-level code that can invalidate properties of the high-level software.

Low-level software security can also be relevant in the context of very different high-level languages, such as the scripting languages embedded within Web pages. For instance, consider a Web application that prompts the user for her name and sends a greeting back to the Web browser. To perform this task, a statement like `response.println("<p>Hello, " + userName + ".</p>");` might be executed by the Web server. This statement may well be written in a high-level programming language that requires explicit declaration of all scripts to be executed in the Web browser. However, scripts are just text strings, and if the string `userName` can be chosen by an attacker, then that attacker may be able to cause arbitrary behavior. The low-level execution in the Web browser will ignore the intent of the higher-level language and execute script code found anywhere, even embedded within `userName` [34].

Therefore, the concepts discussed in this tutorial are applicable more broadly than might appear. Of course, since low-level software security is closely bound to system particulars, the details will be different in contexts other than C and C++ software compiled into machine code. However, in those other contexts there may still be a direct relationship to the attacks and defenses in this tutorial. For instance, defenses based on randomization and secrets much like those in Section 3 have recently been successfully used to prevent low-level attacks on scripting languages in Web applications [26].

1.2 The difficulty of eliminating low-level vulnerabilities

Figure 1 is representative of the attacks and defenses presented in this tutorial. The attacks in Section 2 all exploit vulnerabilities similar to that in Figure 1(a), where a buffer overflow may be possible. For the most part, the defenses in Section 3 use techniques like those in Figure 1(b) and prevent exploits by main-

taining additional information, validating that information with runtime checks, and halting execution if such a check fails.

Unfortunately, unlike in Figure 1, it is often not so straightforward to modify existing source code to use new, safer methods of implementing its functionality. For most code there may not be a direct correspondence between well-known, unsafe library functions and their newer, safer versions. Indeed, existing code can easily be unsafe despite not using any library routines, and vulnerabilities are often obscured by pointer arithmetic or complicated data-structure traversal. (To clarify this point, it is worth comparing the code in Figure 1 with the code in Figure 3, on page 7, where explicit loops implement the same functionality.)

Furthermore, manual attempts to remove software vulnerabilities may give a false sense of security, since they do not always succeed and can sometimes introduce new bugs. For example, a programmer that intends to eliminate buffer overflows in the code of Figure 1(a) might change the `strcpy` and `strcat` function calls as in Figure 1(b), but fail to initialize `t` to be the empty string at the start of the function. In this case, the `strcmp` comparison will be against the unmodified array `t`, if both strings `a` and `b` are longer than `MAX_LEN`.

Thus, a slight omission from Figure 1(b) would leave open the possibility of an exploitable vulnerability as a result of the function reporting that the concatenation of the inputs strings is `"abc"`, even in cases when this is false. In particular, this may occur when, on entry to the function, the array `t` contains `"abc"` as a residual data value from a previous invocation of the function.

Low-level software security vulnerabilities continue to persist due to technical reasons, as well as practical engineering concerns such as the difficulties involved in modifying legacy software. The state of the art in eliminating these vulnerabilities makes use of code review, security testing, and other manual software engineering processes, as well as automatic analyses that can discover vulnerabilities [23]. Furthermore, best practice also acknowledges that some vulnerabilities are likely to remain, and make those vulnerabilities more difficult to exploit by applying defenses like those in this tutorial.

1.3 The assumptions underlying software, attacks, and defenses

Programmers make many assumptions when creating software, both implicitly and explicitly. Some of these assumptions are valid, based on the semantics of the high-level language, as discussed in Section 1.1. For instance, C or C++ programmers may assume that execution does not start at an arbitrary place within a function, but at the start of that function.

Programmers may also make questionable assumptions, such as about the execution environment of their software. For instance, software may be written without concurrency in mind, or in a manner that is dependent on the address encoding in pointers, or on the order of heap allocations. Any such assumptions hinder portability, and may result in incorrect execution when the execution environment changes even slightly.

Finally, programmers may make invalid, mistaken assumptions. For example, in C or C++, programmers may assume that the `int` type behaves like a true,

mathematical integer, or that a memory buffer is large enough for the size of the content it may ever need to hold. All of the above types of assumptions are relevant to low-level software security, and each may make the software vulnerable to attack.

At the same time, attackers also make assumptions, and low-level software attacks rely on a great number of specific properties about the hardware and software architecture of their target. Many of these assumptions involve details about names and the meaning of those names, such as the exact memory addresses of variables or functions and how they are used in the software. These assumptions also relate to the software's execution environment, such as the hardware instruction set architecture and its machine-code semantics.

For example, the Internet Worm of 1988 was successful in large part because of an attack that depended on the particulars of the commonly-deployed VAX hardware architecture, the 4 BSD operating system, and the `fingerd` service. On other systems that were popular at the time, that same attack failed in a manner that only crashed the `fingerd` service, due to the differences in instruction sets and memory layouts [43]. In this manner, attack code is often fragile to the point where even the smallest change prevents the attacker from gaining control, but crashes the target software—effecting a denial-of-service attack.

Defense mechanisms also have assumptions, including assumptions about the capabilities of the attacker, about the likelihood of different types of attacks, about the properties of the software being defended, and about its execution environment. In the attacks and defenses that follow, a note will be made of the assumptions that apply in each case. For instance, of the defenses in Section 3, Defense 1 assumes that attacks make use of a contiguous stack-based buffer overflow, and Defense 4 provides strong guarantees by assuming that Defense 3 is also in place. Also, many defenses (including most of the ones in this tutorial) assume that denial-of-service is not the attacker's goal, and halt the execution of the target software upon the failure of runtime validity checks.

1.4 The presentation of technical details in this tutorial

The presentation in this tutorial paper assumes a basic knowledge of programming languages like C and C++, and their compilation, as might be acquired in an introductory course on compilers. For the most part, relevant technical concepts are introduced when needed. In fact, a large fraction of the technical content is shown in numbered figures whose captions are written to be understandable independent of the main text and without much prior knowledge of low-level software security issues.

As well as giving a number of examples of vulnerable C and C++ software, this tutorial shows many details relating to software execution, such as machine code and execution stack content. Throughout, the details shown will reflect software execution on one particular hardware architecture—a 32-bit x86, such as the IA-32 [11]—but demonstrate properties that also apply to most other hardware platforms.

```
int is_file_foobar( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy( tmp, one );
    strcat( tmp, two );
    return strcmp( tmp, "file://foobar" );
}
```

Fig. 2. A C function that compares the concatenation of two input strings against "file://foobar". This function contains a typical stack-based buffer overflow vulnerability: if the input strings can be chosen by an attacker, then the attacker can direct machine-code execution when the function returns.

The examples show many concrete, hexadecimal values and—in order to avoid confusion—the reader should remember that on the little-endian x86, when four bytes are displayed as a 32-bit integer value, their printed order will be reversed from the order of the bytes in memory. Thus, if the hexadecimal bytes `0xaa`, `0xbb`, `0xcc`, and `0xdd` occur in memory, in that order, then those bytes encode the 32-bit integer `0xddccbbaa`.

2 A selection of low-level attacks on C and C++ software

This section presents four low-level software attacks in full detail and explains how each attack invalidates a property of target software written in the C or C++ languages. The attacks are carefully chosen to be representative of four major classes of attacks: stack-based buffer overflows, heap-based buffer overflows, jump-to-`libc` attacks, and data-only attacks.

No examples are given below of a "format-string attack" or of an "integer-overflow vulnerability". Format-string vulnerabilities are particularly simple to eliminate [12]; therefore, although they have received a great deal of attention in the past, they are no longer a significant, practical concern in well-engineered software. Integer-overflow vulnerabilities [8] do still exist, and are increasingly being exploited, but only as a first step towards attacks like those described below. In this section, Attack 4 is one example where an integer overflow might be the first step in the exploit crafted by the attacker.

As further reading, the survey of Pincus and Baker gives a good general overview of low-level software attacks like those described in this section [38].

2.1 Attack 1: Corruption of a function return address on the stack

It is natural for C and C++ programmers to assume that, if a function is invoked at a particular call site and runs to completion without throwing an exception, then that function will return to the instruction immediately following that same, particular call site.

```
int is_file_foobar_using_loops( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    char* b = tmp;
    for( ; *one != '\0'; ++one, ++b ) *b = *one;
    for( ; *two != '\0'; ++two, ++b ) *b = *two;
    *b = '\0';
    return strcmp( tmp, "file://foobar" );
}
```

Fig. 3. A version of the C function in Figure 2 that copies and concatenates strings using pointer manipulation and explicit loops. This function is also vulnerable to the same stack-based buffer overflow attacks, even though it does not invoke `strcpy` or `strcat` or other C library functions that are known to be difficult to use safely.

Unfortunately, this may not be the case in the presence of software bugs. For example, if the invoked function contains a local array, or buffer, and writes into that buffer are not correctly guarded, then the return address on the stack may be overwritten and corrupted. In particular, this may happen if the software copies to the buffer data whose length is larger than the buffer size, in a *buffer overflow*.

Furthermore, if an attacker controls the data used by the function, then the attacker may be able to trigger such corruption, and change the function return address to an arbitrary value. In this case, when the function returns, the attacker can direct execution to code of their choice and gain full control over subsequent behavior of the software. Figure 2 and Figure 3 show examples of C functions that are vulnerable to this attack.

This attack, sometimes referred to as *return-address clobbering*, is probably the best known exploit of a low-level software security vulnerability; it dates back to before 1988, when it was used in the `fingerd` exploit of the Internet Worm. Indeed, until about a decade ago, this attack was seen by many as the only significant low-level attack on software compiled from C and C++, and "stack-based buffer overflow" were widely considered a synonym for such attacks. More recently, this attack has not been as prominent, in part because other methods of attack have been widely publicized, but also in part because the underlying vulnerabilities that enable return-address clobbering are slowly being eliminated (e.g., through the adoption of newer, safer C library functions).

To give a concrete example of this attack, Figure 4 shows a normal execution stack for the functions in Figures 2 and 3, and Figure 5 shows an execution stack for the same code just after a overflow of the local array—potentially caused by an attacker that can choose the contents of the `two` string provided as input.

Of course, an attacker would choose their input such that the buffer overflow would not caused by "asdfasdfasdfasdf", but another string of bytes. In particular, the attacker might choose `0x48`, `0xff`, and `0x12`, in order, as the final three character bytes of the `two` argument string—and thereby arrange for the func-

```
 address    content
0x0012ff5c 0x00353037 ; argument two pointer
0x0012ff58 0x0035302f ; argument one pointer
0x0012ff54 0x00401263 ; return address
0x0012ff50 0x0012ff7c ; saved base pointer
0x0012ff4c 0x00000072 ; tmp continues 'r' '\0' '\0' '\0'
0x0012ff48 0x61626f6f ; tmp continues 'o' 'o' 'b' 'a'
0x0012ff44 0x662f2f3a ; tmp continues ':' '/' '/' 'f'
0x0012ff40 0x656c6966 ; tmp array:    'f' 'i' 'l' 'e'
```

Fig. 4. A snapshot of an execution stack for the functions in Figures 2 and 3, where the size of the `tmp` array is 16 bytes. This snapshot shows the stack just before executing the `return` statement. Argument `one` is "file://", and argument `two` is "foobar", and the concatenation of those strings fits in the `tmp` array. (Stacks are traditionally displayed with the lowest address at the bottom, as is done here and throughout this tutorial.)

```
 address    content
0x0012ff5c 0x00353037 ; argument two pointer
0x0012ff58 0x0035302f ; argument one pointer
0x0012ff54 0x00666473 ; return address     's' 'd' 'f' '\0'
0x0012ff50 0x61666473 ; saved base pointer 's' 'd' 'f' 'a'
0x0012ff4c 0x61666473 ; tmp continues      's' 'd' 'f' 'a'
0x0012ff48 0x61666473 ; tmp continues      's' 'd' 'f' 'a'
0x0012ff44 0x612f2f3a ; tmp continues      ':' '/' '/' 'a'
0x0012ff40 0x656c6966 ; tmp array:         'f' 'i' 'l' 'e'
```

Fig. 5. An execution-stack snapshot like that in Figure 4, but where argument `one` is "file://" and argument `two` is "asdfasdfasdfasdf". The concatenation of the argument strings has overflowed the `tmp` array and the function return address is now determined by the last few characters of the `two` string.

```
machine code
opcode bytes     assembly-language version of the machine code
  0xcd 0x2e          int 0x2e  ; system call to the operating system
  0xeb 0xfe        L: jmp L    ; a very short, direct infinite loop
```

Fig. 6. The simple attack payload used in this tutorial; in most examples, the attacker's goal will be to execute this machine code. Of these four bytes, the first two are a x86 `int` instruction which performs a system call on some platforms, and the second two are an x86 `jmp` instruction that directly calls itself in an infinite loop. (Note that, in the examples, these bytes will sometimes be printed as the integer `0xfeeb2ecd`, with the apparent reversal a result of x86 little-endianness.)

tion return address to have the value `0x0012ff48`. In this case, as soon as the function returns, the hardware instruction pointer would be placed at the second character of the `two` argument string, and the hardware would start executing the data found there (and chosen by the attacker) as machine code.

In the example under discussion, an attacker would choose their input data so that the machine code for an *attack payload* would be present at address `0x0012ff48`. When the vulnerable function returns, and execution of the attack payload begins, the attacker has gained control of the behavior of the target software. (The attack payload is often called *shellcode*, since a common goal of an attacker is to launch a "shell" command interpreter under their control.)

In Figure 5, the bytes at `0x0012ff48` are those of the second to fifth characters in the string "asdfasdfasdfasdf", namely `'s'`, `'d'`, `'f'`, and `'a'`. When executed as machine code, those bytes do not implement an attack. Instead, as described in Figure 6, an attacker might choose `0xcd`, `0x2e`, `0xeb`, and `0xfe` as a very simple attack payload. Thus, an attacker might call the operating system to enable a dangerous feature, or disable security checks, and avoid detection by keeping the target software running (albeit in a loop).

Return-address clobbering as described above has been a highly successful attack technique—for example, in 2003 it was used to implement the Blaster worm, which affected a majority of Internet users [5]. In the case of Blaster, the vulnerable code was written using explicit loops, much as in Figure 3. (This was one reason why the vulnerability had not been detected and corrected through automatic software analysis tools, or by manual code reviews.)

Attack 1: Constraints and variants

Low-level attacks are typically subject to a number of such constraints, and must be carefully written to be compatible with the vulnerability being exploited.

For example, the attack demonstrated above relies on the hardware being willing to execute the data found on the stack as machine code. However, on some systems the stack is not executable, e.g., because those systems implement the defenses described later in this tutorial. On such systems, an attacker would have to pursue a more indirect attack strategy, such as those described later, in Attacks 3 and 4.

Another important constraint applies to the above buffer-overflow attacks: the attacker-chosen data cannot contain null bytes, or zeros—since such bytes terminate the buffer overflow and prevent further copying onto the stack. This is a common constraint when crafting exploits of buffer overflows, and applies to most of the attacks in this tutorial. It is so common that special tools exist for creating machine code for attack payloads that do not contain any embedded null bytes, newline characters, or other byte sequences that might terminate the buffer overflow (one such tool is Metasploit [18]).

There are a number of attack methods similar to return-address clobbering, in that they exploit stack-based buffer overflow vulnerabilities to target the function-invocation control data on the stack. Most of these variants add a level of indirection to the techniques described above.

One notable attack variant corrupts the base pointer saved on the stack (see Figures 4 and 5) and not the return address sitting above it. In this variant, the vulnerable function may return as expected to its caller function, but, when that caller itself returns, it uses a return address that has been chosen

```
typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
```

Fig. 7. A C function that sets a heap data structure as the concatenation of two input strings, and compares the result against "file://foobar" using the comparison function for that data structure. This function is vulnerable to a heap-based buffer overflow attack if an attacker can choose either or both of the input strings.

by the attacker [30]. Another notable variant of this attack targets C and C++ exception-handler pointers that reside on the stack, and ensures that the buffer overflow causes an exception—at which point a function pointer of the attacker's choice may be executed [32].

2.2 Attack 2: Corruption of function pointers stored in the heap

Software written in C and C++ often combines data buffers and pointers into the same data structures, or objects, with programmers making a natural assumption that the data values do not affect the pointer values. Unfortunately, this may not be the case in the presence of software bugs. In particular, the pointers may be corrupted as a result of an overflow of the data buffer—regardless whether the data structures or objects reside on stack, or in heap memory. Figure 7 shows C code with a function that is vulnerable to such an attack.

To give a concrete example of this attack, Figure 8 shows the contents of the `vulnerable` data structure after the function in Figure 7 has copied data into the `buff` array using the `strcpy` and `strcmp` library functions. Figure 8 shows three instances of the data structure contents: as might occur during normal processing, as might occur in an unintended buffer overflow, and, finally, as might occur during an attack. These instances can occur both when the data structure is allocated on the stack, and also when it is allocated on the heap.

In the last instance of Figure 8, the attacker has chosen the two input strings such that the `cmp` function pointer has become the address of the start of the data structure. At that address, the attacker has arranged for an attack payload to be present. Thus, when the function in Figure 7 executes the `return` statement, and invokes `s->cmp`, it transfers control to the start of the data structure, which contains data of the attacker's choice. In this case, the attack payload is the four

	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0x656c6966	0x662f2f3a	0x61626f6f	0x00000072	0x004013ce

(a) A structure holding "file://foobar" and a pointer to the `strcmp` function.

	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0x656c6966	0x612f2f3a	0x61666473	0x61666473	0x00666473

(b) After a buffer overflow caused by the inputs "file://" and "asdfasdfasdf".

	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0xfeeb2ecd	0x11111111	0x11111111	0x11111111	0x00353068

(c) After a malicious buffer overflow caused by attacker-chosen inputs.

Fig. 8. Three instances of the `vulnerable` data structure pointed to by `s` in Figure 7, where the size of the `buff` array is 16 bytes. Both the address of the structure and its 20 bytes of content are shown. In the first instance, the buffer holds "file://foobar" and `cmp` points to the `strcmp` function. In the second instance, the pointer has been corrupted by a buffer overflow. In the third instance, an attacker has selected the input strings so that the buffer overflow has changed the structure data so that the simple attack payload of Figure 6, page 8, will be executed.

bytes of machine code `0xcd`, `0x2e`, `0xeb`, and `0xfe` described in Figure 6, page 8, and used throughout this tutorial.

It is especially commonplace for C++ code to store object instances on the heap and to combine—within a single object instance—both data buffers that may be overflowed and potentially exploitable pointers. In particular, C++ object instances are likely to contain *vtable pointers*: a form of indirect function pointers that allow dynamic dispatch of virtual member functions. As a result, C++ software may be particularly vulnerable to heap-based attacks.

As a concrete example of vulnerable C++ code, Figure 9 shows the body and supporting classes for a function that performs string concatenation and comparison that should, by now, be familiar.

The C++ code in Figure 9 is vulnerable in much the same manner as the code in Figure 7. In particular, the memory layout of `Vulnerable` objects mirrors that of the `vulnerable` data structure: an array of bytes followed by a pointer to allow for different types of string comparison. However, in this case, the pointer is a not a direct function pointer, but a pointer to the vtables of one of the two `Comparer` classes.

The extra level of indirection to a function pointer requires the attacker to slightly change their attack. As shown in Figure 10, the attacker can place the attack payload four bytes into the overflowed buffer, place the payload address at the start of the buffer, and place in the vtable pointer the address of the start of the buffer. Thereby, the attack payload will be executed when the function in

```
class Comparer
{
public:
    virtual int compare(char* a, char* b) { return stricmp(a,b); }
};

class CaseSensitiveComparer : public Comparer
{
public:
    virtual int compare(char* a, char* b) { return strcmp(a,b); }
};

class Vulnerable
{
    char m_buff[MAX_LEN];
    Comparer m_cmp;
public:
    Vulnerable(Comparer c) : m_cmp(c) {}
    void init(char* str) { strcpy(m_buff, str); }
    void append(char* str) { strcat(m_buff, str); }
    int cmp(char* str) {
        return m_cmp.compare( m_buff, str );
    }
};

int is_file_foobar_using_cpp( Vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    s->init( one );
    s->append( two );
    return s->cmp( "file://foobar" );
}
```

Fig. 9. A C++ version of the code in Figure 7, which uses virtual methods to allow for different types of string comparison. This code is also vulnerable to a heap-based attack, using one level of indirection more than the attack on the code in Figure 7.

	m_buff (char array at start of the object)				m_cmp (vtable)
address:	0x05101010	0x05101014	0x05101018	0x0510101c	0x05101020
content:	0x05101014	0xfeeb2ecd	0x11111111	0x11111111	0x05101010

Fig. 10. The address and contents of a `Vulnerable` object, with a 16 byte buffer, in the case of a buffer overflow attack designed to execute the attack payload from Figure 6. This attack payload is found at address `0x05101014`, four bytes into the buffer.

Figure 9 invokes `s->cmp`, and makes use of the `m_cmp` member of the `Vulnerable` class. (However, note that, in Figure 10, if the object instance were located at

a memory address that contained embedded null bytes, then the attacker would have difficulty writing that address in the buffer overflow.)

Attack 2: Constraints and variants

Heap-based attacks are often constrained by their ability to determine the address of the heap memory that is being corrupted, as can be seen in the examples above. This constraint applies in particular, to all indirect attacks, where a heap-based pointer-to-a-pointer is modified—such as in the C++ attack example above. Furthermore, the exact bytes of those addresses may constrain the attacker, e.g., if the exploited vulnerability is that of a string-based buffer overflow, in which case the address data cannot contain null bytes.

The examples above demonstrate attacks where heap-based buffer overflow vulnerabilities are exploited to corrupt pointers that reside within the same data structure or object as the data buffer that is overflowed. There are two important attack variants, not described above, where heap-based buffer overflows are used to corrupt pointers that reside in other structures or objects, or in the heap metadata.

In the first variant, two data structures or objects reside consecutively in heap memory, the initial one containing a buffer that can be overflowed, and the subsequent one containing a direct, or indirect, function pointer. Heap objects are often adjacent in memory like this when they are functionally related and are allocated in order, one immediately after the other. Whenever these conditions hold, attacks similar to the above examples may be possible, by overflowing the buffer in the first object and overwriting the pointer in the second object.

In the second variant, the attack is based on corrupting the metadata of the heap itself through a heap-based buffer overflow, and exploiting that corruption to write an arbitrary value to an arbitrary location in memory. This is possible because heap implementations contain code such as `p->prev->next = p->next;` to manage doubly-linked lists in their metadata. An attacker that can corrupt the metadata can choose the values of `p->prev` and `p->next`, and thereby choose what is written where. The attacker can then use this capability to write a pointer to the attack payload in the place of any soon-to-be-used function pointer sitting at a known address. For example, in a published attack on the GDI+ JPEG flaw in Windows the attacker overwrote the C++ vtable pointer of a global object whose virtual destructor was invoked as part of error recovery [16].

2.3 Attack 3: Execution of existing code via corrupt pointers

If software does not contain any code for certain functionality—such as performing floating-point calculations, or making system calls to interact with the network—then the programmers may naturally assume that execution of the software will not result in this behavior, or functionality.

Unfortunately, for C or C++ software, this assumption may not hold in the face of bugs and malicious attacks, as demonstrated by attacks like those in this tutorial. As in the previous two examples of attacks, the attacker may be able

```
int median( int* data, int len, void* cmp )
{
    // must have 0 < len <= MAX_INTS
    int tmp[MAX_INTS];
    memcpy( tmp, data, len*sizeof(int) );   // copy the input integers
    qsort( tmp, len, sizeof(int), cmp );    // sort the local copy
    return tmp[len/2];                      // median is in the middle
}
```

Fig. 11. A C function that computes the median of an array of input integers by sorting a local copy of those integers. This function is vulnerable to a stack-based buffer overflow attack, if an attacker can choose the set of input integers.

to cause arbitrary behavior by *direct code injection*: by directly modifying the hardware instruction pointer to execute machine code embedded in attacker-provided input data, instead of the original software. However, there are other means for an attacker to cause software to exhibit arbitrary behavior, and these alternatives can be the preferred mode of attack.

In particular, an attacker may find it preferable to craft attacks that execute the existing machine code of the target software in a manner not intended by its programmers. For example, the attacker may corrupt a function pointer to cause the execution of a library function that is unreachable in the original C or C++ source code written by the programmers—and should therefore, in the compiled software, be never-executed, dead code. Alternatively, the attacker may arrange for reachable, valid machine code to be executed, but in an unexpected order, or with unexpected data arguments.

This class of attacks is typically referred to as *jump-to-`libc`* or *return-to-`libc`* (depending on whether a function pointer or return address is corrupted by the attacker), because the attack often involves directing execution towards machine code in the `libc` standard C library.

Jump-to-`libc` attacks are especially attractive when the target software system is based on an architecture where input data cannot be directly executed as machine code. Such architectures are becoming commonplace with the adoption of the defenses such as those described later in this tutorial. As a result, an increasingly important class of attacks is *indirect code injection*: the selective execution of the target software's existing machine code in a manner that enables attacker-chosen input data to be subsequently executed as machine code. Figure 11 shows a C function that is vulnerable to such an attack.

The function in Figure 11 actually contains a stack-based buffer overflow vulnerability that can be exploited for various attacks, if an attacker is able to choose the number of input integers, and their contents. In particular, attackers can perform return-address clobbering, as described in Attack 1. However, for this particular function, an attacker can also corrupt the comparison-function pointer `cmp` before it is passed to `qsort`. In this case, the attacker can gain control of machine-code execution at the point where `qsort` calls its copy of the

```
...
push   edi             ; push second argument to be compared onto the stack
push   ebx             ; push the first argument onto the stack
call   [esp+comp_fp]   ; call comparison function, indirectly through a pointer
add    esp, 8          ; remove the two arguments from the stack
test   eax, eax        ; check the comparison result
jle    label_lessthan  ; branch on that result
...
```

Fig. 12. Machine code fragment from the `qsort` library function, showing how the comparison operation is called through a function pointer. When `qsort` is invoked in the `median` function of Figure 11, a stack-based buffer overflow attack can make this function pointer hold an arbitrary address.

address	machine code opcode bytes	assembly-language version of the machine code
0x7c971649	0x8b 0xe3	mov esp, ebx ; change the stack location to ebx
0x7c97164b	0x5b	pop ebx ; pop ebx from the new stack
0x7c97164c	0xc3	ret ; return based on the new stack

Fig. 13. Four bytes found within executable memory, in a system library. These bytes encode three machine-code instructions that are useful in the crafting of jump-to-`libc` attacks. In particular, in an attack on the `median` function in Figure 11, these three instructions may be called by the `qsort` code in Figure 12, which will change the stack pointer to the start of the local `tmp` buffer that has been overflowed by the attacker.

corrupted `cmp` argument. Figure 12 shows the machine code in the `qsort` library function where this, potentially-corrupted function pointer is called.

To give a concrete example of a jump-to-`libc` attack, consider the case when the function in Figure 11 is executed on some versions of the Microsoft Windows operating system. On these systems, the `qsort` function is implemented as shown in Figure 12 and the memory address `0x7c971649` holds the four bytes of executable machine code, as shown in Figure 13.

On such a system, the buffer overflow may leave the stack looking like that shown in the "malicious overflow contents" column of Figure 14. Then, when the `qsort` function is called, it is passed a copy of the corrupted `cmp` function-pointer argument, which points to a *trampoline* found within existing, executable machine code. This trampoline is the code found at address `0x7c971649`, which is shown in Figure 13. The effect of calling the trampoline is to, first, set the stack pointer `esp` to the start address of the `tmp` array, (which is held in register `ebx`), second, read a new value for `ebx` from the first integer in the `tmp` array, and, third, perform a return that changes the hardware instruction pointer to the address held in the second integer in the `tmp` array.

The attack subsequently proceeds as follows. The stack is "unwound" one stack frame at a time, as functions return to return addresses. The stack holds data, including return addresses, that has been chosen by the attacker to encode function calls and arguments. As each stack frame is unwound, the return in-

stack address	normal stack contents	benign overflow contents	malicious overflow contents	
0x0012ff38	0x004013e0	0x1111110d	0x7c971649	; `cmp` argument
0x0012ff34	0x00000001	0x1111110c	0x1111110c	; `len` argument
0x0012ff30	0x00353050	0x1111110b	0x1111110b	; `data` argument
0x0012ff2c	0x00401528	0x1111110a	0xfeeb2ecd	; return address
0x0012ff28	0x0012ff4c	0x11111109	0x70000000	; saved base pointer
0x0012ff24	0x00000000	0x11111108	0x70000000	; `tmp` final 4 bytes
0x0012ff20	0x00000000	0x11111107	0x00000040	; `tmp` continues
0x0012ff1c	0x00000000	0x11111106	0x00003000	; `tmp` continues
0x0012ff18	0x00000000	0x11111105	0x00001000	; `tmp` continues
0x0012ff14	0x00000000	0x11111104	0x70000000	; `tmp` continues
0x0012ff10	0x00000000	0x11111103	0x7c80978e	; `tmp` continues
0x0012ff0c	0x00000000	0x11111102	0x7c809a51	; `tmp` continues
0x0012ff08	0x00000000	0x11111101	0x11111101	; `tmp` buffer starts
0x0012ff04	0x00000004	0x00000040	0x00000040	; `memcpy` length argument
0x0012ff00	0x00353050	0x00353050	0x00353050	; `memcpy` source argument
0x0012fefc	0x0012ff08	0x0012ff08	0x0012ff08	; `memcpy` destination arg.

Fig. 14. The address and contents of the stack of the `median` function of Figure 11, where `tmp` is eight integers in size. Three versions of the stack contents are shown, as it would appear just after the call to `memcpy`: a first for input data of the single integer zero, a second for a benign buffer overflow of consecutive integers starting at `0x11111101`, and a third for a malicious jump-to-`libc` attack that corrupts the comparison function pointer to make `qsort` call address `0x7c971649` and the machine code in Figure 13.

```
// call a function to allocate writable, executable memory at 0x70000000
VirtualAlloc(0x70000000, 0x1000, 0x3000, 0x40); // function at 0x7c809a51

// call a function to write the four-byte attack payload to 0x70000000
InterlockedExchange(0x70000000, 0xfeeb2ecd);     // function at 0x7c80978e

// invoke the four bytes of attack payload machine code
((void (*)())0x70000000)();                      // payload at 0x70000000
```

Fig. 15. The jump-to-`libc` attack activity caused by the maliciously-corrupted stack in Figure 14, expressed as C source code. As the corrupted stack is unwound, instead of returning to call sites, the effect is a sequence of function calls, first to functions in the standard Windows library `kernel32.dll`, and then to the attack payload.

struction transfers control to the start of a particular, existing library function, and provides that function with arguments.

Figure 15 shows, as C source code, the sequence of function calls that occur when the stack is unwound. The figure shows both the name and address of the Windows library functions that are invoked, as well as their arguments. The effect of these invocations is to create a new, writable page of executable

memory, to write machine code of the attacker's choice to that page, and to transfer control to that attack payload.

After the trampoline code executes, the hardware instruction pointer address is `0x7c809a51`, which is the start of the Windows library function `VirtualAlloc`, and the address in the stack pointer is `0x0012ff10`, the third integer in the `tmp` array in Figure 14. As a result, when `VirtualAlloc` returns, execution will continue at address `0x7c80978e`, which is the start of the Windows library function `InterlockedExchange`. Finally, the InterlockedExchange function returns to the address `0x70000000`, which at that time holds the attack payload machine code in executable memory.

(This attack is facilitated by two Windows particulars: all Windows processes load the library `kernel32.dll` into their address space, and the Windows calling convention makes library functions responsible for popping their own arguments off the stack. On other systems, the attacker would need to slightly modify the details of the attack.)

Attack 3: Constraints and variants

A major constraint on jump-to-`libc` attacks is that the attackers must craft each such attack with a knowledge of the addresses of the target-software machine code that is useful to the attack. An attacker may have difficulty in reliably determining these addresses, for instance because of variability in the versions of the target software and its libraries, or because of variability in the target software's execution environment. Artificially increasing this variability is a useful defense against many types of such attacks, as discussed later in this tutorial.

Traditionally, jump-to-`libc` attacks have targeted the `system` function in the standard system libraries, which allows the execution of an arbitrary command with arguments, as if typed into a shell command interpreter. This strategy can also be taken in the above attack example, with a few simple changes. However, an attacker may prefer indirect code injection, because it requires launching no new processes or accessing any executable files, both of which may be detected or prevented by system defenses.

For software that may become the target of jump-to-`libc` attacks, one might consider eliminating any fragment of machine code that may be useful to the attacker, such as the trampoline code shown in Figure 13. This can be difficult for many practical reasons. For instance, it is difficult to selectively eliminate fragments of library code while, at the same time, sharing the code memory of dynamic libraries between their instances in different processes; however, eliminating such sharing would multiply the resource requirements of dynamic libraries. Also, it is not easy to remove data constants embedded within executable code, which may form instructions useful to an attacker. (Examples of such data constants include the jump tables of C and C++ `switch` statements.)

Those difficulties are compounded on hardware architectures that use variable-length sequences of opcode bytes for encoding machine-code instructions. For example, on some versions of Windows, the machine code for a system call is encoded using a two-byte opcode sequence, `0xcd`, `0x2e`, while the five-byte sequence

`0x25`, `0xcd`, `0x2e`, `0x00`, and `0x00` corresponds to an arithmetic operation (the operation `and eax, 0x2ecd`, in x86 assembly code). Therefore, if an instruction for this particular `and` operation is present in the target software, then jumping to its second byte can be one way of performing a system call. Similarly, any x86 instruction, including those that read or write memory, may be executed through a jump into the middle of the opcode-byte sequence for some other x86 machine-code instruction.

Indeed, for x86 Linux software, it has been recently demonstrated that it is practical for elaborate jump-to-`libc` attacks to perform arbitrary functionality while executing *only* machine-code found embedded within other instructions [40]. Much as in the above example, these elaborate attacks proceed through the unwinding of the stack, but they may also "rewind" the stack in order to encode loops of activity. However, unlike in the above example, these elaborate attacks may allow the attacker to achieve their goals without adding any new, executable memory or machine code the to target software under attack.

Attacks like these are of great practical concern. For example, the flaw in the `median` function of Figure 11 is in many ways similar to the recently discovered "animated cursor vulnerability" in Windows [22]. Despite existing, deployed defenses, that vulnerability is subject to a jump-to-`libc` attack similar to that in the above example.

2.4 Attack 4: Corruption of data values that determine behavior

Software programmers make many natural assumptions about the integrity of data. As one example, an initialized global variable may be assumed to hold the same, initial value throughout the software's execution, if it is never written by the software. Unfortunately, for C or C++ software, such assumptions may not hold in the presence of software bugs, and this may open the door to malicious attacks that corrupt the data that determine the software's behavior.

Unlike the previous attacks in this tutorial, data corruption may allow the attacker to achieve their goals without diverting the target software from its expected path of machine-code execution—either directly or indirectly. Such attacks are referred to as *data-only*, or *non-control-data*, attacks [10]. In some cases, a single instance of data corruption can be sufficient for an attacker to achieve their goals. Figure 16 shows an example of a C function that is vulnerable to such an attack.

As a concrete example of a data-only attack, consider how the function in Figure 16 makes use of the environment string table by calling the `getenv` routine in the standard C library. This routine returns the string that is passed to another standard routine, `system`, and this string argument determines what external command is launched. An attacker that is able to control the function's two integer inputs is able to write an arbitrary data value to a nearly-arbitrary location in memory. In particular, this attacker is able to corrupt the table of the environment strings to launch an external command of their choice.

Figure 17 gives the details of such an attack on the function in Figure 16, by selectively showing the address and contents of data and code memory. In

```
void run_command_with_argument( pairs* data, int offset, int value )
{
    // must have offset be a valid index into data
    char cmd[MAX_LEN];
    data[offset].argument = value;
    {
        char valuestring[MAX_LEN];
        itoa( value, valuestring, 10 );
        strcpy( cmd, getenv("SAFECOMMAND") );
        strcat( cmd, " " );
        strcat( cmd, valuestring );
    }
    data[offset].result = system( cmd );
}
```

Fig. 16. A C function that launches an external command with an argument value, and stores in a data structure that value and the result of the command. If the offset and value can be chosen by an attacker, then this function is vulnerable to a data-only attack that allows the attacker to launch an arbitrary external command.

```
 address    attack command string data as integers       as characters
0x00354b20  0x45464153 0x4d4d4f43 0x3d444e41 0x2e646d63  SAFECOMMAND=cmd.
0x00354b30  0x20657865 0x2220632f 0x6d726f66 0x632e7461  exe /c "format.c
0x00354b40  0x63206d6f 0x3e20223a 0x00000020             om c:" >

 address    first environment string pointer
0x00353610  0x00353730

 address    first environment string data as integers    as characters
0x00353730  0x554c4c41 0x53524553 0x464f5250 0x3d454c49  ALLUSERSPROFILE=
0x00353740  0x445c3a43 0x6d75636f 0x73746e65 0x646e6120  C:\Documents and
0x00353750  0x74655320 0x676e6974 0x6c415c73 0x7355206c   Settings\All Us
0x00353760  0x00737265                                   ers

 address    opcode bytes    machine code as assembly language
0x004011a1  0x89 0x14 0xc8   mov [eax+ecx*8], edx   ; write edx to eax+ecx*8
```

Fig. 17. Some of the memory contents for an execution of the function in Figure 16, including the machine code for the `data[offset].argument = value;` assignment. If the `data` pointer is `0x004033e0`, the attacker can choose the inputs `offset` = `0x1ffea046` and `value` = `0x00354b20`, and thereby make the assignment instruction change the first environment string pointer to the "format" command string at the top.

this case, before the attack, the environment string table is an array of pointers starting at address `0x00353610`. The first pointer in that table is shown in Figure 17, as are its contents: a string that gives a path to the "all users profile". In a correct execution of the function, some other pointer in the environment string

table would be to a string, such as `SAFECOMMAND=safecmd.exe`, that determines a safe, external command to be launched by the `system` library routine.

However, before reading the command string to launch, the machine-code assignment instruction shown in Figure 17 is executed. By choosing the `offset` and `value` inputs to the function, the attacker can make `ecx` and `edx` hold arbitrary values. Therefore, the attacker can make the assignment write any value to nearly any address in memory, given knowledge of the `data` pointer. If the `data` pointer is `0x004033e0`, then that address plus $8*$`0x1ffea046` is `0x00353610`, the address of the first environment string pointer. Thus, the attacker is able to write the address of their chosen attack command string, `0x00354b20`, at that location. Then, when `getenv` is called, it will look no further than the first pointer in the environment string table, and return a command string that, when launched, may delete data on the "C:" drive of the target system.

Several things are noteworthy about this data-only attack and the function in Figure 16. First, note that there are multiple vulnerabilities that may allow the attacker to choose the `offset` integer input, ranging from stack-based and heap-based buffer overflows, through integer overflow errors, to a simple programmer mistake that omitted any bounds check. Second, note that although `0x1ffea046` is a positive integer, it effectively becomes negative when multiplied by eight, and the assignment instruction writes to an address before the start of the `data` array. Finally, note that this attack succeeds even when the table of environment strings is initialized before the execution starts, and the table is never modified by the target software—and when the table should therefore logically be read-only given the semantics of the target software.

Attack 4: Constraints and variants

There are two major constraints on data-only attacks. First, the vulnerabilities in the target software are likely to allow only certain data, or a certain amount of data to be corrupted, and potentially only in certain ways. For instance, as in the above example, a vulnerability might allow the attacker to change a single, arbitrary four-byte integer in memory to a value of their choice. (Such vulnerabilities exist in some heap implementations, as described on page 13; there, an arbitrary write is possible through the corruption of heap metadata, most likely caused by the overflow of a buffer stored in the heap. Many real-world attacks have exploited this vulnerability, including the GDI+ JPEG attack in Windows [10, 16].)

Second, even when an attacker can replace any amount of data with arbitrary values, and that data may be located anywhere, a data-only attack will be constrained by the behavior of the target software when given arbitrary input. For example, if the target software is an arithmetic calculator, a data-only attack might only be able to cause an incorrect result to be computed. However, if the target software embeds any form of an interpreter that performs potentially dangerous operations, then a data-only attack could control the input to that interpreter—allowing the attacker to perform the dangerous operations. The `system` standard library routine is an example of such an interpreter; many

applications, such as Web browsers and document viewers, embed other interpreters for scripting languages.

To date, data-only attacks have not been prominent. Rather, data corruption has been most frequently utilized as one step in other types of attacks, such as direct code injection, or an jump-to-`libc` attack. This may change with the increased deployment of defenses, including the defenses described below.

3 Defenses That Preserve High-level Language Properties

This section presents, in detail, six effective, practical defenses against low-level software attacks on x86 machine-code software, and explains how each defense is based on preserving a property of target software written in the C or C++ languages. These defenses are stack canaries, reordering of stack variables, non-executable data, control-flow integrity, encrypted pointers, and address-space layout randomization. They have been selected based on their efficiency, and ease-of-adoption, as well as their effectiveness.

In particular, this section describes neither defenses based on instruction-set randomization [27], nor defenses based on dynamic information flow tracking, or tainting, or other forms of data-flow integrity enforcement [9, 36]. Such techniques can offer strong defenses against all the attacks in Section 2, although, like the defenses below, they also have limitations and counterattacks. However, these defenses have drawbacks that make their deployment difficult in practice.

For example, unless they are supported by specialized hardware, they incur significant overheads. On unmodified, commodity x86 hardware, defenses based on data-flow integrity may double the memory requirements, and may make execution up to 37 times slower [36]. Because these defenses also double the number of memory accesses, even the most heavily optimized mechanism is still likely to run software twice as slow [9]. Such overheads are likely to be unacceptable in many scenarios, e.g., for server workloads where a proportional increase in cost may be expected. Therefore, in practice, these defenses may never see widespread adoption—especially since equally good protection may be achievable using a combination of the below defenses.

This section does not attempt a comprehensive survey of the literature on these defenses. However, related material can be found with a search based on the papers referenced in this section, and their discussion of other work.

3.1 Defense 1: Checking Stack Canaries on Return Addresses

The C and C++ languages do not specify how function return addresses are represented in stack memory. Rather, these, and many other programming languages, hold abstract most elements of a function's invocation stack frame in order to allow for portability between hardware architectures and to give compilers flexibility in choosing an efficient low-level representation. This flexibility enables an effective defense against some attacks, such as the return-address clobbering of Attack 1.

```
 address    content
0x0012ff5c 0x00353037 ; argument two pointer
0x0012ff58 0x0035302f ; argument one pointer
0x0012ff54 0x00401263 ; return address
0x0012ff50 0x0012ff7c ; saved base pointer
0x0012ff4c 0x00000000 ; all-zero canary
0x0012ff48 0x00000072 ; tmp continues 'r' '\0' '\0' '\0'
0x0012ff44 0x61626f6f ; tmp continues 'o' 'o' 'b' 'a'
0x0012ff40 0x662f2f3a ; tmp continues ':' '/' '/' 'f'
0x0012ff3c 0x656c6966 ; tmp array:     'f' 'i' 'l' 'e'
```

Fig. 18. A stack snapshot like that shown in Figures 4 where a "canary value" has been placed between the `tmp` array and the saved base pointer and return address. Before returning from functions with vulnerabilities like those in Attack 1, it is an effective defense to check that the canary is still zero: an overflow of a zero-terminated string across the canary's stack location will not leave the canary as zero.

In particular, on function calls, instead of storing return addresses directly onto the stack, C and C++ compilers are free to generate code that stores return addresses in an encrypted and signed form, using a local, secret key. Then, before each function return, the compiler could emit code to decrypt and validate the integrity of the return address about to be used. In this case, assuming that strong cryptography is used, an attacker that did not know the key would be unable to cause the target software to return to an address of their choice as a result of a stack corruption—even when the target software contains an exploitable buffer overflow vulnerability that allows such corruption.

In practice, it is desirable to implement an approximation of the above defense, and get most of the benefits without incurring the overwhelming cost of executing cryptography code on each function call and return.

One such approximation requires no secret, but places a public *canary* value right above function-local stack buffers. This value is designed to warn of dangerous stack corruption, much as a coal-mine canary would warn about dangerous air conditions. Figure 18 shows an example of a stack with an all-zero canary value. Validating the integrity of this canary is an effective means of ensuring that the saved base pointer and function return address have not been corrupted—given the assumption that attacks are only possible through stack corruption based on the overflow of a string buffer. For improved defenses, this public canary may contain other bytes, such as newline characters, that frequently terminate the copying responsible for string-based buffer overflows. For example, some implementations have used the value `0x000aff0d` as the canary [14].

Stack-canary defenses may be improved by including in the canary value some bits that should be unknown to the attacker. For instance, this may help defend against return-address clobbering with an integer overflow, such as is enabled by the `memcpy` vulnerability in Figure 11. Therefore, some implementations of stack canary defenses, such as Microsoft's /GS compiler option [7], are based on a random value, or *cookie*.

```
function_with_gs_check:
     ; function preamble machine code
     push ebp                          ; save old base pointer on the stack
     mov  ebp, esp                     ; establish the new base pointer
     sub  esp, 0x14                    ; grow the stack for buffer and cookie
     mov  eax, [__security_cookie]     ; read cookie value into eax
     xor  eax, ebp                     ; xor base pointer into cookie
     mov  [ebp-4], eax                 ; write cookie above the buffer
     ...
     ; function body machine code
     ...
     ; function postamble machine code
     mov  ecx, [ebp-4]                 ; read cookie from stack, into ecx
     xor  ecx, ebp                     ; xor base pointer out of cookie
     call __security_check_cookie      ; check ecx is cookie value
     mov  esp, ebp                     ; shrink the stack back
     pop  ebp                          ; restore old, saved base pointer
     ret                               ; return

__security_check_cookie:
     cmp  ecx, [__security_cookie]     ; compare ecx and cookie value
     jnz  ERR                          ; if not equal, goto an error handler
     ret                               ; else return
ERR: jmp __report_gsfailure            ; report failure and halt execution
```

Fig. 19. The machine code for a function with a local array in a fixed-size, 16-byte stack buffer, when compiled using the Windows /GS implementation of stack cookies in the most recent version of the Microsoft C compiler [7, 24]. The canary is a random cookie value, combined with the base pointer. In case the local stack buffer is overflowed, this canary is placed on the stack above the stack buffer, just below the return address and saved base pointer, and checked before either of those values are used.

Figure 19 shows the machine code for a function compiled with Microsoft's /GS option. The function preamble and postamble each have three new instructions that set and check the canary, respectively. With /GS, the canary placed on the stack is a combination of the function's base pointer and the function's *module cookie*. Module cookies are generated dynamically for each process, using good sources of randomness (although some of those sources are observable to an attacker running code on the same system). Separate, fresh module cookies are used for the executable and each dynamic library within a process address space (each has its own copy of the `__security_cookie` variable in Figure 19). As a result, in a stack with multiple canary values, each will be unique, with more dissimilarity where the stack crosses module boundaries.

Defense 1: Overhead, Limitations, Variants, and Counterattacks

The is little enforcement overhead from stack canary defenses, since they are only required in functions with local stack buffers that may be overflowed. (An

overflow in a function does not affect the invocation stack frames of functions it calls, which are lower on the stack; that function's canary will be checked before any use of stack frames that are higher on the stack, and which may have been corrupted by the overflow.) For most C and C++ software this overhead amounts to a few percent [14, 15]. Even so, most implementations aim to reduce this overhead even further, by only initializing and checking stack canaries in functions that contain a local string `char` array, or meet other heuristic requirements. As a result, this defense is not always applied where it might be useful—as evidenced by the recent ANI vulnerability in Windows [22].

Stack canaries can be an efficient and effective defense against Attack 1, where the attacker corrupts function-invocation control data on the stack. However, stack canaries only check for corruption at function exit. Thus, they offer no defense against Attacks 2, 3, and 4, which are based on corruption of the heap, function-pointer arguments, or global data pointers.

Stack canaries are a widely deployed defense mechanism. In addition to Microsoft's /GS, StackGuard [14] and ProPolice [15] are two other notable implementations. Given its simple nature, it is somewhat surprising that there is significant variation between the implementations of this defense, and these implementations have varied over time [7, 21]. In part, this reflects the ongoing arms race between attackers and defenders; however, it is also because stack-canaries have been combined with other defenses, such as Defense 2 below.

Stack canary defenses are subject to a a number of counterattacks. Most notably, even when the only exploitable vulnerability is a stack-based buffer overflow, the attackers may be able to craft an attack that is not based on return-address clobbering. For example, the attack may corrupt a local variable, an argument, or some other value that is used before the function exits.

Also, the attacker may attempt to guess, or learn the cookie values, which can lead to a successful attack given enough luck or determination. The success of this counterattack will depend on the exploited vulnerability, the attacker's access to the target system, and the particulars of the target software. (For example, if stack canaries are based on random cookies, then the attacker may be able to exploit certain format-string vulnerabilities to learn which canary values to embed in the data of the buffer overflow.)

3.2 Defense 2: Moving function-local variables below stack buffers

Most details about the function-invocation stack frame are left unspecified in the C and C++ languages, to give flexibility in the compilation of those language aspects down to a low-level representation. In particular, the compiler is free to lay out function-local variables in any order on the stack, and to generate code that operates not on function arguments, but on copies of those arguments. This flexibility enables an efficient defense against attacks based on stack corruption, such as Attacks 1 and 3.

In this defense, the compiler places arrays and other function-local buffers above all other function-local variables on the stack. Also, the compiler makes copies of function arguments into new, function-local variables that also sit below

stack address	stack contents	overflow contents	
0x0012ff38	0x004013e0	0x1111110d	; `cmp` argument
0x0012ff34	0x00000001	0x1111110c	; `len` argument
0x0012ff30	0x00353050	0x1111110b	; `data` argument
0x0012ff2c	0x00401528	0x1111110a	; return address
0x0012ff28	0x0012ff4c	0x11111109	; saved base pointer
0x0012ff24	0x00000000	0x11111108	; `tmp` final 4 bytes
0x0012ff20	0x00000000	0x11111107	; `tmp` continues
0x0012ff1c	0x00000000	0x11111106	; `tmp` continues
0x0012ff18	0x00000000	0x11111105	; `tmp` continues
0x0012ff14	0x00000000	0x11111104	; `tmp` continues
0x0012ff10	0x00000000	0x11111103	; `tmp` continues
0x0012ff0c	0x00000000	0x11111102	; `tmp` continues
0x0012ff08	0x00000000	0x11111101	; `tmp` buffer starts
0x0012ff04	0x004013e0	0x004013e0	; *local copy of `cmp` argument*
0x0012ff00	0x00000004	0x00000040	; `memcpy` length argument
0x0012fefc	0x00353050	0x00353050	; `memcpy` source argument
0x0012fef8	0x0012ff08	0x0012ff08	; `memcpy` destination argument

Fig. 20. A version of the stack shown in Figure 14 for the `median` function of Figure 11, showing the contents for both valid inputs and in the case of a benign buffer overflow. A local copy of the `cmp` argument has been placed at address `0x0012ff04`, below `tmp`; therefore, this copy cannot be corrupted by an overflow of the `tmp` buffer. If the code in the body of the function makes use of this copy, not the original argument, then this prevents attacks based on the vulnerability exploited by Attack 3.

any buffers in the function. As a result, these variables and arguments are not subject to corruption through an overflow of those buffers. Figure 20 shows an example of how this defense might prevent the stack-based exploit in Attack 3. In this example, a local copy has been made of the `cmp` function-pointer argument corrupted by the attack; since this local copy resides below the buffer on the stack, it cannot be corrupted in an overflow of that buffer.

Defense 2: Overhead, Limitations, Variants, and Counterattacks

Placing local variables and copies of function arguments below any buffers in a function's stack frame has negligible enforcement overheads: reordering local variables has effectively zero overhead, and the overhead of local argument copies is also close to zero. Furthermore, this overhead applies only to functions with local stack buffers that may be overflowed. As a result, in practice, the overhead of this defense is often too small to be reliably measured [15].

This defense is both efficient and effective, but it is also limited. In particular, this defense offers no protection against Attacks 2 and 4 or other attacks that do not exploit stack-based buffer overflow vulnerabilities.

However, this defense does not completely prevent an attacker from exploiting the effects of a stack-based buffer overflow. For instance, although local variables or arguments may not be corrupted, these variables may contain pointers into

a region of the stack that may possibly be corrupted by the attacker. Thus, in the case of a string pointer function argument, the attacker may not be able to change the address in the pointer, but they will be able to change the contents of the string itself—as long as the buffer overflow reaches up to the location on the stack where the string resides.

This defense is combined with stack canaries in most implementations, such as ProPolice and Microsoft's /GS. This is a good fit, since both techniques defend against the effects of stack-based buffer overflow, but only stack canaries offer a means of detecting when stack corruption may have occurred. There is some variation between these implementations. In particular, in order to further reduce enforcement overhead, not all function arguments may be copied, but only code and data pointers and other arguments that meet some heuristic requirements. (Of course, this may permit some attacks that might otherwise not be possible.)

There are few counterattacks to this defense. An attacker may attempt to craft a more indirect stack-based buffer overflow exploit that corrupts the contents of pointers into the stack. However, the attacker is also likely to turn their attention from attacks based on stack-based buffer overflow to other means of attack, such as those described in Attacks 2 and 4.

3.3 Defense 3: Making data not be executable as machine code

Many high-level languages allow code and data to reside in two, distinct types of memory. The C and C++ languages follow this tradition, and do not specify what happens when code pointers are read and written as data, or what happens when a data pointer is invoked as if it were a function pointer. This under-specification brings important benefits to the portability of C and C++ software, since it must sometimes run on systems where code and data memory are truly different. It also enables a particularly simple and efficient defense against direct-code-injection exploits, such as those in Attacks 1 and 2.

If data memory is not executable, then Attacks 1 and 2 fail as soon as the hardware instruction pointer reaches the first byte of the attack payload (e.g., the bytes `0xfeeb2ecd` described in Figure 6, and used throughout this tutorial). Even when the attacker manages to control the flow of execution, they cannot simply make control proceed directly to their attack payload. This is a simple, useful barrier to attack, which can be directly applied to most software, since, in practice, most software never treats data as code.

(Some legacy software will execute data as a matter of course; other software uses self-modifying code and writes to code memory as a part of regular, valid execution. For example, this behavior can be seen in some efficient, just-in-time interpreters. However, such software can be treated as a special case, since it is uncommon and increasingly rare.)

Defense 3: Overhead, Limitations, Variants, and Counterattacks

In its implementation on modern x86 systems, non-executable data has some performance impact because it relies on double-size, extended page tables. The

NX page-table-entry bit, which flags memory as non-executable, is only found in PAE page tables, which are double the size of normal tables, and are otherwise not commonly used. The precise details of page-table entries can significantly impact the overall system performance, since page tables are a frequently-consulted part of the memory hierarchy—with thousands of lookups a second and, in some cases, a lookup every few instructions. However, for most workloads, the overhead should be in the small percents, and will often be close to zero.

Non-executable data defends against direct code injection attacks, but offers no barrier to exploits such as those in Attacks 3 and 4. For any given direct code-injection attack, it is likely that an attacker can craft an indirect jump-to-`libc` variant, or a data-only exploit [10]. Thus—although this defense can be highly useful when used in combination with other defenses—by itself, it is not much of a stumbling block for attackers.

On Microsoft Windows, and most other platforms, software will typically execute in a mode where writing to code memory generates a hardware exception. In the past, some systems have also generated such an exception when the hardware instruction pointer is directed to data memory, i.e., upon an attempt to execute data as code. However, until recently, commodity x86 hardware has only supported such exceptions through the use of segmented memory—which runs counter to the flat memory model that is fundamental to most modern operating systems. (Despite being awkward, x86 segments have been used to implement non-executable memory, e.g., stacks, but these implementations are limited, for instance in their support for multi-threading and dynamic libraries.)

Since 2003, and Windows XP SP2, commodity operating systems have come to support the x86 extended page tables where any given memory page may be marked as non-executable, and x86 vendors have shipped processors with the required hardware support. Thus, it is now the norm for data memory to be non-executable, in particular when running the Windows operating system.

On most legacy x86 processors that use unmodified page tables, it is actually possible to make data pages be non-executable using a clever technique first implemented in the PaX project [37]. This technique builds on the fact that, at the very top of the memory hierarchy on those x86 processors, code and data is separated into two distinct memories: the code *i-cache* and data *d-cache*. By maintaining an invariant, this technique can ensure that data memory is never present in the i-cache, and therefore that data memory is never executed. Although ingenious, this technique does not work on all x86 processors; it can also have significant performance overhead and has a race condition on multi-processor systems. Therefore, it never saw significant adoption before being superseded by the current x86 hardware support for non-executable data pages.

(The technique works as follows: all data pages in the page table are marked as invalid, thereby preventing data memory from entering either of the two top-level memories. Rather, an access to a data memory address will cause a page-fault handler to be invoked; this handler can load the data memory into the d-cache only, by briefly marking as valid the page-table entry for the data

memory and reading the data memory, e.g., using the `mov` instruction. As a result, until it is evicted from the d-cache, the memory will be accessible as data. However, because its page-table entry remains marked as invalid, the memory will never enter the i-cache, and therefore never be executable.)

Indirect code injection, jump-to-`libc` attacks, and data-only attacks are all effective counterattacks to this defense Even so, non-executable data can play a key role in an overall defense strategy; for instance, when combined with Defense 6 below, this defense can prevent an attacker from knowing the location of any executable memory bytes that could be useful to an attack.

3.4 Defense 4: Enforcing control-flow integrity on code execution

As in all high-level languages, it is not possible for software written in the C and C++ languages to perform arbitrary control-flow transfers between any two points in its code. Compared to the exclusion of data from being executed as code, the policies on control-flow between code are much more fine-grained

For example, the behavior of function calls is only defined when the callee code is the start of a function—even when the caller invokes that code through a function pointer. Also, it is not valid to place a label into an expression, and `goto` to that label, or otherwise transfer control into the middle of an expression being evaluated. Transferring control into the middle of a machine code instruction is certainly not a valid, defined operation, in any high-level language—even though the hardware may allow this, and this may be useful to an attacker (see Attack 3, page 18).

Furthermore, within the control flow that a language permits in general, only a small fraction will, in fact, be possible in the semantics of a particular piece of software written in that language. For most software, control flow is either completely static (e.g., as in a C `goto` statement), or allows only a small number of possibilities during execution.

For example, consider the case when the entire high-level language software consists of the C++ code of Figure 9 on page 12. There, in the `cmp` function, it is only possible for the invocation of `m_cmp.compare` to pass control to the start of a `compare` member function in one of the two `Comparer` classes. Also, it is clear that when either of those `compare` functions returns, it can only be to the call site in the `cmp` function. Similarly, for all C or C++ software, any indirect control transfers, such as through function pointers or at return statements, will have only a small number of valid targets.

Dynamic checks can ensure that the execution of low-level software does not stray from a restricted set of possibilities allowed by the high-level software. The runtime enforcement of such a Control-Flow Integrity, or CFI, security policy is a highly effective defense against low-level software attacks [2, 3].

As a concrete example of the benefits of CFI enforcement, consider the published attack on the GDI+ JPEG flaw in Windows [16]. This attack starts by causing a heap memory corruption that results in a global variable being overwritten; this variable holds a C++ object pointer. When this pointer is later used for calling a virtual destructor, the attacker has the possibility of executing

```
class Vulnerable
{
    char m_buff[MAX_LEN];
    Comparer m_cmp;
public:
    Vulnerable(Comparer c) : m_cmp(c) {}
    // ... elided code ...
    int cmp(char* str) {
        if( (m_cmp.compare == &Comparer::compare) ||
            (m_cmp.compare == &CaseSensitiveComparer::compare) )
        {
            return m_cmp.compare( m_buff, str );
        }
        else throw report_memory_corruption_error();
    }
};
```

Fig. 21. An excerpt of the C++ code in Figure 9 with explicit CFI checks that only allow valid comparison methods to be invoked at runtime—thereby preventing the exploit in Attack 2. (This code may not be accepted by modern C++ compilers, because of how it reads the address of a virtual member function in an object instance.)

```
int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // ... elided code ...
    if( (s->cmp == strcmp) || (s->cmp == stricmp) ) {
        return s->cmp( s->buff, "file://foobar" );
    } else {
        return report_memory_corruption_error();
    }
}
```

Fig. 22. An excerpt of the C code in Figure 7 with explicit CFI checks that only allow the proper comparison methods to be invoked at runtime—assuming only `strcmp` and `stricmp` are possible. These CFI checks prevent the exploit on this function in Attack 2.

code of their choice, much as in the C++ example in Attack 2. A CFI check at this callsite can prevent this exploit, for instance by restricting valid destinations to the C++ virtual destructor methods of the GDI+ library.

There are several strategies possible in the implementation of CFI enforcement. For instance, CFI may be enforced by dynamic checks that compare the target address of each computed control-flow transfer to a set of allowed destination addresses. Such a comparison may be performed by the machine-code equivalent of a switch statement over a set of constant addresses. Programmers can even make CFI checks explicitly in their software, as shown in Figures 21 and 22. However, unlike in Figures 21 and 22, it is not possible to write software that explicitly performs CFI checks on return addresses, or other inaccessible

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

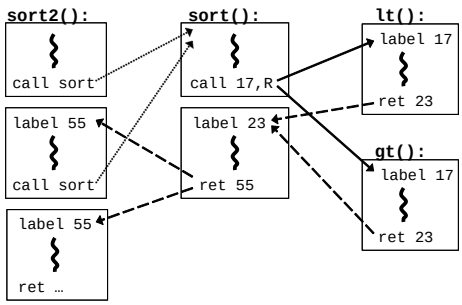

Fig. 23. Three C functions and an outline of their possible control flow, as well as how an CFI enforcement mechanism based on CFI labels might apply to the functions. In the outline, the CFI labels `55`, `17`, and `23` are found at the valid destinations of computed control-flow instructions; each such instruction is also annotated with a CFI label that corresponds to its valid destinations.

pointers; for these, CFI checks must be added by the compiler, or some other mechanism. Also, since the set of allowed destination addresses may be large, any such sequence of explicit comparisons is likely to lead to unacceptable overhead.

One efficient CFI enforcement mechanism, described in [2], modifies according to a given control-flow graph, both the *source* and *destination* instructions of computed control-flow transfers. Two destinations are *equivalent*, when the CFG contains edges to each from the same set of sources. At each destination, a *CFI label* is inserted, that identifies equivalent destinations, i.e., destinations with the same set of possible sources. The CFI labels embed a value, or bit pattern, that distinguishes each; these values need not be secret. Before each source instruction, a dynamic CFI check is inserted that ensures that the runtime destination has the proper CFI label.

Figure 23 shows a C program fragment demonstrating this CFI enforcement mechanism. In this figure, a function `sort2` calls a `qsort`-like function `sort` twice, first with `lt` and then with `gt` as the pointer to the comparison function. The right side of Figure 23 shows an outline of the machine-code blocks for these four functions and all control-flow-graph edges between them. In the figure, edges for direct calls are drawn as light, dotted arrows; edges from source instructions are drawn as solid arrows, and return edges as dashed arrows. In this example, `sort` can return to two different places in `sort2`. Therefore, there are two CFI labels in the body of `sort2`, and an CFI check when returning from `sort`, using `55` as the CFI label. (Note that CFI enforcement does not guarantee to which of the two callsites `sort` must return; for this, other defenses, such as Defense 1, must be employed.)

Also, in Figure 23, because `sort` can call either `lt` or `gt`, both comparison functions start with the CFI label `17`, and the `call` instruction, which uses a function pointer in register `R`, performs a CFI check for `17`. Finally, the CFI label `23` identifies the block that follows the comparison callsite in `sort`, so both comparison functions return with a CFI check for `23`.

machine-code opcode bytes	machine code in assembly
...	...
0x57	push edi
0x53	push ebx
0x8b 0x44 0x24 0x24	mov eax, [esp+comp_fp]
0x81 0x78 0xfc 0x78 0x56 0x34 0x12	cmp [eax-0x4], 0x12345678
0x75 0x13	jne cfi_error_label
0xff 0xd0	call eax
0x0f 0x18 0x80 0xdd 0xcc 0xbb 0xaa	prefetchnta [0xaabbccdd]
0x83 0xc4 0x08	add esp, 0x8
0x85 0xc0	test eax, eax
0x7e 0x02	jle label_lessthan
...	...

Fig. 24. A version of Figure 12, showing how CFI checks as in [2] can be added to the `qsort` library function where it calls the comparison function pointer. Before calling the pointer, it is placed in a register `eax`, and a comparison establishes that the four bytes `0x12345678` are found immediately before the destination code, otherwise execution goes to a security error. After the call instruction, an executable, side-effect-free instruction embeds the constant `0xaabbccdd`; by comparing against this constant, the comparison function can establish that it is returning to a valid call site.

Figure 24 shows a concrete example of how CFI enforcement based on CFI labels can look, in the case of x86 machine-code software. Here, the CFI label `0x12345678` identifies all comparison routines that may be invoked by `qsort`, and the CFI label `0xaabbccdd` identifies all of their valid call sites. This style of CFI enforcement has good performance, and also gives strong guarantees. By choosing the bytes of CFI labels carefully, so they don't overlap with code, even an attacker that controls all of data memory cannot divert execution from the permitted control-flow graph—assuming that data is also non-executable.

The CFI security policy dictates that software execution must follow a path of a control-flow graph, determined ahead of time, that represents all possible valid executions of the software. This graph can be defined by analysis—source-code analysis, binary analysis, or execution profiling. This graph does not need to be perfectly accurate, but needs only be a conservative approximation of the control-flow graph possible in the software, as written in its high-level programming language. To be conservative, the graph must err on the side of allowing all valid executions of the software, even this may entail allowing some invalid executions as well. For instance, the graph might conservatively permit the start of a few-too-many functions as the valid destinations of a source instruction where a function pointer is invoked.

Defense 4: Overhead, Limitations, Variants, and Counterattacks

CFI enforcement incurs only modest overhead. With the CFI enforcement mechanism in [2], which instruments x86 machine code much as is shown in Figure 24, the reported code-size increase is around 8%, and execution slowdown ranges

from 0% to 45% on a set of processor benchmarks, with a mean of 16% Even so, this overhead is significant enough that CFI enforcement has, to date, seen only limited adoption. However, a form of CFI is enforced by the Windows SafeSEH mechanism, which limits dispatching of exceptions to a set of statically-declared exception handlers; this mechanism does not incur measurable overheads.

CFI enforcement offers no protection against Attack 4 or other data-only attacks. However, CFI can be an highly effective defense against all attacks based on controlling machine-code execution, including Attacks 1, 2, and 3.

In particular, CFI enforcement is likely to prevent all variants of Attack 3, i.e., jump-to-`libc` attacks that employ trampolines or opportunistic executable byte sequences such as those found embedded within machine-code instructions. This is the case even if CFI enforces only a coarse-grained approximation of the software control-flow graph, such as allowing function-pointer calls to the start of any function with the same argument types, and allowing functions to return to any of their possible call sites [2].

CFI enforcement mechanisms vary both in their mechanisms and in their policy. Some mechanisms establish the validity of each computed control transfer by querying a separate, static data structure, which can be a hash table, a bit vector, or a structure similar to multi-level page tables [42]. Other mechanisms execute the software in a fast machine-code interpreter that enforces CFI on control flow [29]. Finally, a coarse-grained form of CFI can be enforced by making all computed-control-flow destinations be aligned on multi-word boundaries. (However, in this last case, any "basic block" is effectively a valid destination, so trampolines and elaborate jump-to-`libc` attacks are still feasible.) The complexity and overheads of these CFI mechanisms varies, but is typically greater than that described above, based on CFI labels.

In a system with CFI enforcement, any exploit that does not involve controlling machine-code execution is a likely counterattack; this includes not only data-only attacks, such as Attack 4, but also other, higher-level attacks, such as social engineering and flaws in programming interfaces [4]. In addition, depending on the granularity of CFI enforcement policy, and how it is used in combination with other defenses, there may still exist possibilities for certain jump-to-`libc` attacks, for instance where a function is made to return to a dynamically-incorrect, but statically-possible, call site.

3.5 Defense 5: Encrypting addresses in code and data pointers

The C and C++ languages do not exactly specify how pointers encode memory addresses, since this representation is highly platform specific. Just as C and C++ hold return addresses abstract, which allows Defense 1, other pointers are only subject to a few, well-defined operations that hold abstract their address value. Therefore, one promising line of defense is to encrypt the addresses stored in data and code pointers using a secret key, unknown to the attacker.

All of the attacks in Section 2 depend on pointer corruption in some way. As long as all pointers contain encrypted addresses—including direct and indirect function pointers, return addresses, base pointers, and pointers to data—then

```
           buff (char array at start of the struct)     cmp
address: 0x00353068 0x0035306c 0x00353070 0x00353074 0x00353078
content: 0x656c6966 0x662f2f3a 0x61626f6f 0x00000072 0xbd6ba903
```

Fig. 25. A version of the data structure in Figure 8(a). The structure holds the string "file://foobar" and a pointer to the `strcmp` function, whose address is `0x004013ce`. In this example, the address has been explicitly encrypted using the Windows `EncodePointer` library function, and is stored as `0xbd6ba903` in the data structure.

this defense can thwart all of these attacks. For instance, encrypted pointers can prevent Attack 4, where a pointer in the environment string table is corrupted. When crafting this attack, if the attacker does not know the pointer encoding, then they cannot know what integer value to write as an address in the environment string table.

Unfortunately, in practice, software is most commonly written for expediency and efficiency, not portability. Thus, existing C and C++ software often contains elaborate pointer arithmetic, and operates on pointers cast into integral types in ways that make it difficult to adopt defenses based on encrypted pointers. This is in stark difference from the other defenses in this section: it is far more common for software to rely on the address encoding in pointers than, say, for software to make use of self-modifying code, or execute data as code. As a result, defense mechanisms such as PointGuard [13], which attempt to pervasively and automatically apply encrypted pointers, have seen very limited adoption.

However, if encrypted pointers are used explicitly by programmers, then the programmers can ensure that no pointer arithmetic is invalidated by the encryption. Such explicit, selective application of encrypted pointers is a useful defense for security-critical code that is a likely target of attacks. Figure 25 shows the contents of a data structure where a function pointer has been explicitly encrypted using a library function available on Windows.

This defense has seen significant adoption in security-conscious software systems. For instance, in recent versions of the Windows heap implementation, many heap metadata pointers are explicitly encrypted to prevent certain forms of heap-based attacks discussed on page 13, in Attack 2

Figure 26 shows how, on Windows, programmers can use pointer encoding to make the C++ function in Figure 9 less vulnerable to attack. In this code fragment, Windows library routines are explicitly invoked to encrypt and decrypt the address in the comparison pointer, using a per-process random value. Thus this address will always appear encrypted in the object instance contents, requiring the attacker to guess or learn the key in order to perform attacks such as that shown in Figure 10.

Defense 5: Overhead, Limitations, Variants, and Counterattacks

The performance effects of this defense will vary based on how pervasively pointers are encrypted, and what encryption method is used. To keep overheads low, only a weak form of encryption is typically used, e.g., xor-ing pointers with a

```
class LessVulnerable
{
    char m_buff[MAX_LEN];
    void* m_cmpptr;
public:
    LessVulnerable(Comparer* c) {
        m_cmpptr = EncodePointer( c );
    }
    // ... elided code ...
    int cmp(char* str) {
        Comparer* mcmp;
        mcmp = (Comparer*) DecodePointer( m_cmpptr );
        return mcmp->compare( m_buff, str );
    }
};
```

Fig. 26. An excerpt of a variant of the C++ code in Figure 9 where the comparison pointer is encoded using a random value specific to the process executing the software. This code invokes library routines for the encoding and decoding; these particular routines are present on Windows XP SP2 and more recent versions of Microsoft Windows.

secret value. When encryption is simply an inline xor operation, its performance effects will be limited to a few percent, even when applied widely [13]. In Windows, encryption is based on an xor operation and a bit-wise rotation, using either per-process or system-wide random values established using good sources of randomness [20]; one system call is performed when using a per-process secret. As applied in Windows utilities and systems software, this defense has enforcement overheads that are small enough to be hard to measure for most workloads.

The main limitation of this defense is that encrypted pointers must be selectively applied to existing software, due to their potential incompatibility with pointer arithmetic. Also, in the common case where encryption does not include any signature or integrity check, this defense may not detect attempted attacks.

Many variants of this defense are possible, depending on which pointers are encrypted. In particular, the encoding of addresses held in pointers does not need to be the same for all pointers in a given piece of software. Instead, pointers could be assigned into equivalence classes, or "colors", and each color could be given a different encoding, as long as no instruction that accessed a pointer made use of more than one color. This variant can make defenses more fine grained and, if applied pervasively to all or most pointers, can approximate the benefits of other, general constraints on the software's data flow [9].

The counterattacks to this defense depend on which pointers are encrypted, and, of course, attacks that do not involve pointers are still possible. In particular, this defense may not prevent attacks based on corrupting the contents of data, such as a buffer overflow of a boolean value that signifies successful authentication [10]. Such attacks do not corrupt pointers, but still require the attacker to know, in some form, the location of the data contents to be corrupted.

3.6 Defense 6: Randomizing the layout of code and data in memory

The C and C++ languages specify neither where code is located in memory, nor the location of variables, arrays, structures, or objects. For software compiled from these languages, the layout of code and data in memory is decided by the compiler and execution environment. This layout directly determines all concrete addresses used during execution—and attacks, including all of the attacks in Section 2, typically depend on these concrete addresses.

Therefore, a simple, pervasive form of address encryption can be achieved by shuffling, or randomizing, the layout of software in the memory address space, in a manner that is unknown to the attacker. Defenses based on such Address-Space Layout Randomization, or ASLR, can be a highly practical, effective barrier against low-level attacks. Such defenses were first implemented in the PaX project [37] and have recently been deployed in Windows Vista [19, 24].

ASLR defenses can be used to change the addresses of all code, global variables, stack variables, arrays, and structures, objects, and heap allocations; with ASLR those addresses are derived from a random value, chosen for the software being executed and the system on which it executes. These addresses, and the memory-layout shuffling, may be public information on the system where the software executes. However, low-level software attacks—including most worms, viruses, adware, spyware, and malware—are often performed by remote attackers that have no existing means of running code on their target system, or otherwise inspect the addresses utilized on that system. To overcome ASLR defenses, such attackers will have to craft attacks that do not depend on addresses, or somehow guess or learn those addresses.

ASLR is not intended to defend against attackers that are able to control the software execution, even to a very small degree. Like many other defenses that rely on secrets, ASLR is easily circumvented by an attacker that can read the software's memory. Once an attacker is able to execute even the smallest amount of code of their choice (e.g., in a jump-to-`libc` attack), it should be safely assumed that the attacker can read memory and, in particular, that ASLR is no longer an obstacle. Fortunately, ASLR and the other defenses in this tutorial can be highly effective in preventing attackers from successfully executing even a single machine-code instruction of their choice.

As a concrete example of ASLR, Figure 27 shows two execution stacks for the `median` function of Figure 11, taken from two executions of that function on Windows Vista, which implements ASLR defenses. These stacks contain code addresses, including a function pointer and return address; they also include addresses in data pointers that point into the stack, and in the `data` argument which points into the heap. All of these addresses are different in the two executions; only the integer inputs remain the same.

On many software platforms, ASLR can be applied automatically, in manner that is compatible even with legacy software. In particular, unlike Defense 5, ASLR changes only the concrete values of addresses, not how those addresses are encoded in pointers; this makes ASLR compatible with common, legacy programming practices that depend on the encoding of addresses.

stack one		stack two		
address	contents	address	contents	
0x0022feac	0x008a13e0	0x0013f750	0x00b113e0	; `cmp` argument
0x0022fea8	0x00000001	0x0013f74c	0x00000001	; `len` argument
0x0022fea4	0x00a91147	0x0013f748	0x00191147	; `data` argument
0x0022fea0	0x008a1528	0x0013f744	0x00b11528	; return address
0x0022fe9c	0x0022fec8	0x0013f740	0x0013f76c	; saved base pointer
0x0022fe98	0x00000000	0x0013f73c	0x00000000	; `tmp` final 4 bytes
0x0022fe94	0x00000000	0x0013f738	0x00000000	; `tmp` continues
0x0022fe90	0x00000000	0x0013f734	0x00000000	; `tmp` continues
0x0022fe8c	0x00000000	0x0013f730	0x00000000	; `tmp` continues
0x0022fe88	0x00000000	0x0013f72c	0x00000000	; `tmp` continues
0x0022fe84	0x00000000	0x0013f728	0x00000000	; `tmp` continues
0x0022fe80	0x00000000	0x0013f724	0x00000000	; `tmp` continues
0x0022fe7c	0x00000000	0x0013f720	0x00000000	; `tmp` buffer starts
0x0022fe78	0x00000004	0x0013f71c	0x00000004	; `memcpy` length argument
0x0022fe74	0x00a91147	0x0013f718	0x00191147	; `memcpy` source argument
0x0022fe70	0x0022fe8c	0x0013f714	0x0013f730	; `memcpy` destination arg.

Fig. 27. The addresses and contents of the stacks of two different executions of the same software, given the same input. The software is the `median` function of Figure 11, the input is an array of the single integer zero, and the stacks are snapshots taken at the same point as in Figure 14. The snapshots are taken from two executions of that function on Windows Vista, with a system restart between the executions. As a result of ASLR defenses, only the input data remains the same in the two executions. All addresses are different; even so, some address bits remain the same since, for efficiency and compatibility with existing software, ASLR is applied only at a coarse granularity.

However, ASLR is both easier to implement, and is more compatible with legacy software, when data and code is shuffled at a rather coarse granularity. For instance, software may simultaneously use more than a million heap allocations; however, on a 32-bit system, if an ASLR mechanism randomly spread those allocations uniformly throughout the address space, then only small contiguous memory regions would remain free. Then, if that software tried to allocate an array whose size is a few tens of kilobytes, that allocation would most likely fail—even though, without this ASLR mechanism, it might certainly have succeeded. On the other hand, without causing incompatibility with legacy software, an ASLR mechanism could change the base address of all heap allocations, and otherwise leave the heap implementation unchanged. (This also avoids triggering latent bugs, such as the software's continued use of heap memory after deallocation, which are another potential source of incompatibility.)

In the implementation of ASLR on Windows Vista, the compilers and the execution environment have been modified to avoid obstacles faced by other implementations, such as those in the PaX project [37]. In particular, the software executables and libraries of all operating system components and utilities have been compiled with information that allows their relocation in memory at load time. When the operating system starts, the system libraries are located

sequentially in memory, in the order they are needed, at a starting point chosen randomly from 256 possibilities; thus a jump-to-`libc` attack that targets the concrete address of a library function will have less than a 0.5% chance of succeeding. This randomization of system libraries applies to all software that executes on the Vista operating system; the next time the system restarts, the libraries are located from a new random starting point.

When a Windows Vista process is launched, several other addresses are chosen randomly for that process instance, if the main executable opts in to ASLR defenses. For instance, the base of the initial heap is chosen from 32 possibilities. The stacks of process threads are randomized further: the stack base is chosen from 32 possibilities, and a pad of unused memory, whose size is random, is placed on top of the stack, for a total of about 16 thousand possibilities for the address of the initial stack frame. In addition, the location of some other memory regions is also chosen randomly from 32 possibilities, including thread control data and the process environment data (which includes the table corrupted in Attack 4). For processes, the ASLR implementation chooses new random starting points each time that a process instance is launched.

An ASLR implementation could be designed to shuffle the memory layout at a finer granularity than is done in Windows Vista. For instance, a pad of unused memory could be inserted within the stack frame of all (or some) functions; also, the inner memory allocation strategy of the heap could be randomized. However, in Windows Vista, such an ASLR implementation would incur greater overhead, would cause more software compatibility issues, and might be likely to thwart mostly attacks that are already covered by other deployed defenses. In particular, there can be little to gain from shuffling the system libraries independently for each process instance [41]—and such an ASLR implementation would be certain to cause large performance and resource overheads.

Defense 6: Overhead, Limitations, Variants, and Counterattacks

The enforcement overhead of ASLR defenses will vary greatly depending on the implementation. In particular, implementations where shared libraries may be placed at different addresses in different processes will incur greater overhead and consume more memory resources.

However, in its Windows Vista implementation, ASLR may actually slightly improve performance. This improvement is a result of ASLR causing library code to be placed contiguously into the address space, in the order that the code is actually used. This encourages a tight packing of frequently-used page table entries, which has performance benefits (cf. the page-table changes for non-executable data, discussed on page 27).

ASLR can provide effective defenses against all of the attacks in Section 2 of this tutorial, because it applies to the addresses of both code and data. Even so, as discussed on page 34 for Defense 5, some data-only attacks remain possible, where the attacks do not depend on concrete addresses, but rely on corrupting the contents of the data being processed by the target software.

The more serious limitation of ASLR is the small number of memory layout shuffles that are possible on commodity 32-bit hardware—especially given the coarse shuffling granularity that is required for efficiency and compatibility with existing software. As a result, ASLR creates only at most a few thousand possibilities that an attacker must consider, and any given attack will be successful against a significant (albeit small) number of target systems. The number of possible shuffles in an ASLR implementation can be greatly increased on 64-bit platforms, which are starting to be adopted. However, current 64-bit hardware is limited to 48 usable bits and can therefore offer at most a 64-thousand-fold increase in the number of shuffles possible [44].

Furthermore, at least on 32-bit systems, the number of possible ASLR shuffles is insufficient to provide a defense against scenarios where the attacker is able to retry their attack repeatedly, with new addresses [41]. Such attacks are realistic. For example, because a failed attack did not crash the software in the case of the recent ANI vulnerability in Windows [22], an attack, such as a script in a malicious Web page, could try multiple addresses until a successful exploit was found. However, in the normal case, when failed attacks crash the target software, attacks based on retrying can be mitigated by limiting the number of times the software is restarted. In the ASLR implementation in Windows Vista, such limits are in place for many system components.

ASLR defenses provide one form of software diversity, which has been long known to provide security benefits. One way to achieve software diversity is to deploy multiple, different implementations of the same functionality. However, this approach is costly and may offer limited benefits: its total cost is proportional to the number of implementations and programmers are known to make the same mistakes when implementing the same functionality [33].

A more attractive defense—which can offer more diversity, at little cost—is to artificially perturb some of the low-level properties of existing, deployed implementations [17]. ASLR is one, relatively coarse-grained variant of this defense. Other, finer-grained variants exist, including techniques based on automatically creating multiple software versions through randomized obfuscation of the high-level software specification [39]. While preserving the software's high-level semantics, such obfuscation can change the semantics as well as the addresses of low-level code and data. However, unlike ASLR, defenses based on finer-grained diversity have many costs, including performance overheads and increases to the cost of software-engineering processes such as testing and debugging.

ASLR has a few counterattacks other than the data-only, content-based attacks, and the persistent guessing of an attacker, which are both discussed above. In particular, an otherwise harmless information-disclosure vulnerability may allow an attacker to learn how addresses are shuffled, and circumvent ASLR defenses. Although unlikely, such a vulnerability may be present because of a format-string bug, or because the contents of uninitialized memory are sent on the network when that memory contains residual addresses.

Another type of counterattack to ASLR defenses is based on overwriting only the low-order bits of addresses, which are predictable because ASLR is applied at

a coarse granularity. Such overwrites are sometimes possible through buffer overflows on little-endian architectures, such as the x86. For example, in Figure 27, if there were useful trampoline machine-code to be found seven bytes into the `cmp` function, then changing the least-significant byte of the `cmp` address on the stack from `0xe0` to `0xe7` would cause that code to be invoked. An attacker that succeeded in such corruption might well be able to perform a jump-to-`libc` attack much like that in Attack 3. (However, for this particular stack, the attacker would not succeed, since the `cmp` address will always be overwritten completely when the vulnerability in the `median` function in Figure 11 is exploited.)

Despite the above counterattacks, ASLR is an effective barrier to attack, especially when combined with the defenses described previously in this section. Indeed, with such a combination of defenses, an attacker may be most likely to counter with a higher-level attack, such as one based on higher-level interfaces such as Web scripting languages, or simply based on social engineering.

4 Summary and discussion

The distinguishing characteristic of low-level software attacks is that they are dependent on the low-level details of the software's executable representation and its execution environment. As a result, defenses against such attacks can be based on changing those details in ways that are compatible with the software's specification in a higher-level programming language.

As in Defense 1, integrity bits can be added to the low-level representation of state, to make attacks more likely to be detected, and stopped. As in Defense 2, the low-level representation can be reordered and replicated to move it away from corruption possibilities. As in Defenses 3 and 4, the low-level representation can be augmented with a conservative model of behavior and with runtime checks that ensure execution conforms to that model. Finally, as in Defenses 1, 5, and 6, the low-level representation can be encoded with a secret that the attacker must guess, or otherwise learn, in order to craft functional attacks.

However, defenses like those in this tutorial fall far short of a guarantee that the software exhibits only the low-level behavior that is possible in the software's higher-level specification. Such guarantees are hard to come by. For languages like C and C++, there are efforts to build certifying compilers that can provide such guarantees, for correct software [6, 31]. Unfortunately, even these compilers offer few, or no guarantees in the presence of bugs, such as buffer-overflow vulnerabilities. Many of these bugs can be eliminated by using other, advanced compiler techniques, like those used in the Cyclone [25], CCured [35], and Deputy [45] systems. But these techniques are not widely applicable: they require pervasive source-code changes, runtime memory-management support, restrictions on concurrency, and result in significant enforcement overhead.

In comparison, the defenses in this tutorial have very low overheads, require no source code changes but at most re-compilation, and are widely applicable to legacy software written in C, C++, and similar languages. For instance, they have been applied pervasively to recent Microsoft software, including all the com-

	Attack 1	Attack 2	Attack 3	Attack 4
Defense 1	Partial defense		Partial defense	Partial defense
Defense 2	Partial defense		Partial defense	Partial defense
Defense 3	Partial defense	Partial defense	Partial defense	
Defense 4	Partial defense	Partial defense	Partial defense	
Defense 5		Partial defense	Partial defense	Partial defense
Defense 6	Partial defense	Partial defense	Partial defense	Partial defense

Table 1. A table of the relationship between the attacks and defenses in this tutorial. None of the defenses completely prevent the attacks, in all of their variants. The first two defenses apply only to the stack, and are not an obstacle to the heap-based Attack 2. Defenses 3 and 4 apply only to the control flow of machine-code execution, and do not prevent the data-only Attack 4. Defense 5 applies only to pointers that programmers can explicitly encode and decode; thus, it cannot prevent the return-address clobbering in Attack 1. When combined with each other, the defenses are stronger than when they are applied in isolation.

ponents of the Windows Vista operating system. As in that case, these defenses are best used as one part of a comprehensive software-engineering methodology designed to to reduce security vulnerabilities. Such a methodology should include, at least, threat analysis, design and code reviews for security, security testing, automatic analysis for vulnerabilities, and the rewriting of software to use safer languages, interfaces, and programming practices [23].

The combination of the defenses in this tutorial forms a substantial, effective barrier to all low-level attacks—although, as summarized in Table 1, each offers only partial protection against certain attacks. In particular, they greatly reduce the likelihood that an attacker can exploit a low-level security vulnerability for purposes other than a denial-of-service attack. Because these defenses are both effective, and easy to adopt in practice, in the next few years, they are likely to be deployed for most software. Their adoption, along with efforts to eliminate buffer overflows and other underlying security vulnerabilities, offers some hope that, for C and C++ software, low-level software security may become less of a concern in the future.

Acknowledgments Thanks to Martín Abadi for suggesting the structure of this tutorial, and to Yinglian Xie for proofreading and for suggesting useful improvements to the exposition.

References

1. M. Abadi. Protection in programming-language translations. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 868–883. Springer-Verlag, 1998. Also Digital Equipment Corporation Systems Research Center report No. 154, April 1998.
2. M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, implementations, and applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2005. Also as Microsoft Research Technical Report MSR-TR-05-18, February 2005.
3. M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Engineering Methods*, 2005. Also as Microsoft Research Technical Report MSR-TR-05-17, May 2005.
4. R.J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001.
5. M. Bailey, E. Cooke, F. Jahanian, D. Watson, and J. Nazario. The Blaster worm: Then and now. *IEEE Security and Privacy*, 03(4):26–31, 2005.
6. S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer-Verlag, 2006.
7. B. Bray. Compiler security checks in depth, 2002. `http://msdn2.microsoft.com/en-us/library/aa290051(vs.71).aspx`.
8. D. Brumley, T c. Chiueh, R. Johnson, H. Lin, and D. Song. Efficient and accurate detection of integer-based attacks. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'07)*, February 2007.
9. M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 11–11, Berkeley, CA, USA, 2006. USENIX Association.
10. S. Chen, J. Xu, E.C. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the Usenix Security Symposium*, pages 177–192, 2005.
11. Intel Corporation. Intel IA-32 architecture, software developer's manual, Volumes 1–3, 2007. `http://developer.intel.com/design/Pentium4/documentation.htm`.
12. C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the Usenix Security Symposium*, 2001.
13. C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the Usenix Security Symposium*, pages 91–104, 2003.
14. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Usenix Security Symposium*, pages 63–78, 1998.
15. H. Etoh and K. Yoda. ProPolice—improved stack smashing attack detection. *IPSJ SIGNotes Computer Security (CSEC)*, 14, October 2001.
16. E. Florio. GDIPLUS VULN - MS04-028 - CRASH TEST JPEG. *full-disclosure at lists.netsys.com*, 2004. Forum message, sent September 15.

17. S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *HOTOS '97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 67, Washington, DC, USA, 1997. IEEE Computer Society.
18. J.C. Foster. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress Publishing, 2007.
19. M. Howard. Alleged bugs in Windows Vistas ASLR implementation, 2006. `http://blogs.msdn.com/michael_howard/archive/2006/10/04/Alleged-Bugs-in-Windows-Vista_1920_s-ASLR-Implementation.aspx`.
20. M. Howard. Protecting against pointer subterfuge (redux), 2006. `http://blogs.msdn.com/michael_howard/archive/2006/08/16/702707.aspx`.
21. M. Howard. Hardening stack-based buffer overrun detection in VC++ 2005 SP1, 2007. `http://blogs.msdn.com/michael_howard/archive/2007/04/03/hardening-stack-based-buffer-overrun-detection-in-vc-2005-sp1.aspx`.
22. M. Howard. Lessons learned from the animated cursor security bug, 2007. `http://blogs.msdn.com/sdl/archive/2007/04/26/lessons-learned-from-the-animated-cursor-security-bug.aspx`.
23. M. Howard and S. Lipner. *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, USA, 2006.
24. M. Howard and M. Thomlinson. Windows Vista ISV security, April 2007. `http://msdn2.microsoft.com/en-us/library/bb430720.aspx`.
25. T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the Usenix Technical Conference*, pages 275–288, 2002.
26. M. Johns and C. Beyerlein. SMask: Preventing injection attacks in Web applications by approximating automatic data/code separation. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 284–291, New York, NY, USA, 2007. ACM Press.
27. G. S. Kc, A.D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, New York, NY, USA, 2003. ACM Press.
28. A. Kennedy. Securing the .NET programming model. *To appear in special issue of Theoretical Computer Science*, 2007. Earlier version presented at APPSEM II Workshop, in Munich, Germany, September 12-15, 2005.
29. V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the Usenix Security Symposium*, pages 191–206, 2002.
30. Klog. The frame pointer overwrite. *Phrack*, 9(55), 1999.
31. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.
32. D. Litchfield. Defeating the stack buffer overflow prevention mechanism of Microsoft Windows 2003 Server, 2003. `http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf`.
33. B. Littlewood, P. Popov, and L. Strigini. Modeling software design diversity: a review. *ACM Comput. Surv.*, 33(2):177–208, 2001.
34. B. Livshits and Ú. Erlingsson. Using Web application construction frameworks to protect against code injection attacks. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 95–104, New York, NY, USA, 2007. ACM Press.

35. G.C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 128–139, 2002.
36. J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'07)*, February 2005.
37. PaX Project. The PaX project, 2004. `http://pax.grsecurity.net/`.
38. J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.
39. R. Pucella and F.B. Schneider. Independence from obfuscation: A semantic framework for diversity. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 230–241, Washington, DC, USA, 2006. IEEE Computer Society. Expanded version available as Cornell University Computer Science Department Technical Report TR 2006-2016.
40. H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86), 2006. In submission, `http://hovav.net/dist/geometry.pdf`.
41. H. Shacham, M. Page, B. Pfaff, E-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM Press.
42. C. Small. A tool for constructing safe extensible C++ systems. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, 1997.
43. E.H. Spafford. The Internet worm program: An analysis. *SIGCOMM Comput. Commun. Rev.*, 19(1):17–57, 1989.
44. Wikipedia. x86-64, 2007. `http://en.wikipedia.org/wiki/X86-64`.
45. F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G.C. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 4–4, Berkeley, CA, USA, 2006. USENIX Association.