Mitigations for Low-Level Coding Vulnerabilities: Incomparability and Limitations

Jonathan Pincus and Brandon Baker
Microsoft Research
{jpincus, babaker}@microsoft.com

Abstract

Exploits of vulnerabilities due to low-level coding defects such as buffer overruns and integer overflows are a major source of security problems. Mitigation techniques attempt to limit damage from these vulnerabilities. While many such techniques have been developed and deployed, work to date has proceeded in a haphazard fashion. A more structured approach to the problem requires the understanding between vulnerabilities, exploits, and mitigations. Multiple exploit techniques can apply to any individual vulnerability, and mitigations focus either on the underlying defects or directly on specific exploits. We reduce all published exploits to combinations of three primitive techniques, and provide a taxonomy of mitigation techniques. Using these taxonomies, we show that mitigation techniques are incomparable: each category prevents some exploits not addressed by other. No combination of currently-deployed mitigation techniques defeats all currently-known exploits.

1 Introduction

Security vulnerabilities related to low-level coding defects such as buffer overruns and integer overflows account for the largest share of CERT advisories as well as high-profile worms from the original Internet Worm through Blaster. When a vulnerabilitiy is discovered, malicious crackers devise *exploits* that take advantage of the vulnerability to attack a system. In parallel, software providers issue patches, which remove the vulnerability by fixing the underlying defect. Systems are protected if the patch is installed before they are attacked. As a way of providing additional protection to un-patched systems, software providers are increasingly deploying technologies that attempt to *mitigate* the effect of such vulnerabilities – typically by reducing the consequences of an attack from “elevation of privilege” to “denial of service”. While many such *mitigation techniques* have been developed, there are known attacks on all mitigations.

Attackers exploit these vulnerabilities by controlling the value in one or more memory locations and thus modifying the program’s behavior. All published exploits reduce to sequences of one or more *exploit primitives*: modifying an instruction, modifying a data value, or modifying an address. The popular *code injection* style of exploits, for example, involves at least two primitives: insertion of a sequence of instructions into the program's memory, followed by modification of an address so that control is transferred to the newly-inserted instructions. Variations may involve additional steps, for example modifying an address to a subsequent assignment to modify an additional memory location.

One category of mitigation techniques attempts to prevent (or, alternatively, to detect) defects. Techniques focusing on defects are effective against all possible exploits of the defect. Other categories of mitigation techniques, by contrast, attempt to prevent or detect specific exploit primitives. A final category attempts to detect execution of injected code.

A key observation is that multiple exploit techniques are possible for most security defects. As a result, it is often easy for attackers to defeat exploit-focused mitigations.

All widely-used categories of mitigation techniques are “incomparable”: for any two categories of mitigations M_1 and M_2, there are at least some exploits prevented by M_1 that are not prevented by M_2, and vice versa. The combination of multiple mitigation techniques is a natural response to this incomparability; and indeed, some obviously-synergistic techniques are frequently combined in practice. Unfortunately, broadly-deployed combinations of mitigations still fail to prevent some exploit techniques. Such incomplete combinations still leave the system vulnerable to exploitation.

The major contributions of this paper are

- Identification of three primitive exploit techniques which underlie all published exploits of low-level coding defects
- A taxonomy of mitigation techniques that distinguishes between exploit- and defect-focused mitigations.
- A demonstration of the incomparability of existing mitigation techniques.
- A demonstration that no combination of currently-deployed mitigation techniques stops all known exploit techniques.

Section 2 and 3 of the paper discuss exploits and mitigations, respectively. Section 4 discusses the incomparability of mitigations, and section 5 discusses the effectiveness of mitigation techniques. We close with discussions of future and related work.

2 Exploits

An *exploit* is the means by which an attacker can take advantage of a vulnerability. An individual exploit is an instance of a more general *exploit technique*. An exploit can then be coupled with a *payload* (often referred to as a shellcode) to create malicious code.

In this paper, we focus on exploit techniques for vulnerabilities caused by low-level C/C++ coding defects such as buffer over- and under-runs, integer overflows and size mismatches, format string bugs, and double frees (see [Hogl] for definitions and examples of these defects).

An attacker can exploit such a vulnerability to achieve an elevation of privilege by controlling the values in one or more memory locations L at a point in the program's execution P. Table 1 informally describes some L/P pairs that are particularly popular. The attacker can achieve this control via a sequence of one or more *primitive operations* which control the contents of memory locations. We further describe primitives in terms of what data is controlled: *instructions*, *data values,* or *addresses*. For example, the exploit used by the Internet Worm reduces to two such operations: storing a sequence of instructions (in a stack buffer), and then modifying the saved return address to point to that buffer. When program execution reached the return instruction, control was transferred to the stack buffer, and then the new instructions were executed.

The most straightforward approach to controlling the value in a memory location is for the attacker to modify it. Exploits for uninitialized variables, race conditions and usages of freed memory also typically involve alternative approaches to control. For uninitialized variables, for example, the attacker must somehow influence the initial ("garbage") value of the uninitialized location.

The exploit becomes effective at program point P, but operations modifying values in L typically occur at some earlier program point P'. For the Internet Worm exploit, P' is where the buffer is overrun (and the values in L are set), and P is when the return instruction is actually executed. More complex exploits may rely on setting multiple locations at multiple program points. If program execution prevents P from being reached, or modifies the values in L, the exploit will not succeed.

Behavior of the program may limit the attacker's control over the value(s) being placed in L. In some cases, input may be filtered or transformed; in other cases, the defect itself may restrict the attacker's control (for example, to a maximum number of bytes). We ignore this additional complexity in this paper.

Although most real-world exploits involve multiple operations, it is instructive to consider the individual primitives in more detail.

Table 1: Popular location/program point pairs to control

Location	Program Point	Reason
Boolean (or bit) used to determine whether security checks must be provided	when the value is tested	disable security check
"checksum" or "cookie" value	before the value is tested	defeat self-checking code
non-const pointer	before an assignment via that pointer	use the assignment to control some other location
Return address (stored on stack on x86)	before the function returns	change program flow by returning to an unexpected location (e.g., code supplied by the attacker)
Function pointer	before a call through that pointer	change program flow
vtbl pointer	before a virtual function call	change the computation of which function to call, thus changing program flow

2.1 Instructions

Modifying a program's instructions can obviously change its behavior. For example, changing a jne instruction to a jeq effectively reverses the meaning of a comparison; if the comparison relates to (for example) checking passwords for validity, such a change to an instruction could allow access to any user with an invalid password.

[Hogl] briefly mentions such an exploit: modifying an opcode for a jmp instruction stored as part of a "jump table". Direct applications of this technique are typically infeasible because of an inherent mitigation provided in operating systems and chip architectures commonly in use today: the instruction stream of a program is typically non-writable after it is loaded.

2.2 Data values

A powerful, but tricky, form of exploit involves controlling a data value that is used in a security-critical decision. A classic 1992 NFS exploit [XFor] of an sized integer mismatch involved passing a UID (user ID) in which the low 16 bits were zero: the combination of an initial check of UID != 0 (instead of the correct (unsigned short)UID == 0) allowed the attacker root access. [Hogl] illustrates the potential consequences of such an exploit: changing a single bit in the memory of a running Microsoft Windows system can disable all access checking.

Such an exploit is straightforward if a security-critical value is directly derived from attacker input, as in the NFS example. Single-operation data-modification exploits are also possible if a security-critical variable (local, static, global, or field of a struct) happens to be adjacent to a buffer that can be overrun. These may well be relatively rare situations, however, as there have been no such published exploits.

2.3 Addresses

An exploit's control of addresses serves one of three purposes, depending on how the address is used in subsequent program execution.

If the address is used as the target of a transfer of control (for example, a saved return address or a function pointer address), then controlling the address effectively changes the program's control-flow graph representation. This is often used to transfer control to a sequence of attacker-controlled instructions (typically provided by another primitive operation), or to existing code in the program with attacker-controlled data (again, typically provided by another primitive operation).

If the address is used as an lvalue in a subsequent assignment, then controlling the address effectively allows the attacker to accomplish an "arbitrary memory write" and modify other memory locations. The first step in *heap smashing* exploits, for example, modifies pointers used in the internal data structures of dynamic memory managers such as **malloc**. As a result, a subsequent call to **free** on the memory whose pointer has been corrupted modifies any four bytes of memory at a location chosen by the attacker.

More generally, control of an address that is used as a pointer allows the attacker to control the program's access to the pointed-to locations. For example, modifying an address used as a string pointer effectively gives the attacker control over the string itself (without modifying the originally-pointed-to memory locations).

These latter two techniques combine effectively, and are thus important for building up more complicated exploits. The arbitrary memory write in a heap smashing exploit, for example, can be used to modify an address used in a subsequent control-flow transfer, which in turn leads to execution of instructions inserted by the attacker. *Vtable hijacking* exploits use initial control over an address used as a pointer (to the virtual function table in a C++ object) to exert indirect control over an address used as a control-flow target (a virtual function pointer – that is, an entry in the table).

Most exploit techniques involve the attacker modifying all the bytes of an address (typically four or eight bytes, depending on the underlying machine architecture). In a *partial overwrite*, by contrast, the attacker modifies only a subset of the bytes of an existing address – typically, the one or two least significant bytes. This effectively allows the attacker to create a new address relative to the original address, as opposed to an absolute address that is possible with full control. Partial overwrites thus provide an important counter to certain randomization-based mitigation techniques, a topic discussed in more detail below.

2.4 Complex exploits

Exploits in the real world typically involve multiple primitive operations. Some of the most important categories of exploits include *CFG modification* (which in turn decomposes into *code injection* and *arc injection* exploits), *pure data exploits*, and *multi-defect* exploits.

CFG modification includes any technique in which the attacker inserts a new edge and optionally block into the program's control-flow graph. In *arc injection* exploits, the attacker only inserts an edge, inducing a new control transfer to a block already in

Figure 2: simplified version of MS03-026 vulnerability

```
extern int global_magic_number;
void vulnerable(char *input, size_t max_length) {
      char buff[MAX_SIZE];
      int *p;
      int i;
      int stack_canary;

P0:   stack_canary = global_magic_number;
P1:   memset(buff, input, max_length);  // buffer overrun!
      ...
P2:   *p = i;
P3:   if (stack_canary != global_magic_number)
            // buffer overrun has occurred!
            exit();
P:    return;
}
```

a program, often calling an existing function with arguments containing data controlled by the attacker. In *code injection* exploits, by contrast, the attacker inserts a new block as well, by first providing a set of instructions that are later executed., and additionally modifying some address so that execution is transferred to those instructions.

Obviously, all code injection attacks involve at least two primitive operations. The Internet Worm introduced *stack smashing,* the most basic form of code injection technique (described above); other variants involve modification of other addresses (function pointers, exception blocks, frame pointers) and usage of heap memory rather than stack memory for the instructions, or an initial integer overflow as in a recent Linux kernel vulnerability [CERT].

A difficulty with code injection techniques in general is for the exploit to know the absolute address A where the instructions are stored. *Trampolining* is a clever solution to this in situations where some register R is known to have an address relative to A. In this case, the exploit transfers to a (known) absolute location that has a jump or call instruction via R, which indirectly causes control to be transferred to A. Blaster and Slammer both used exploits involving stack smashing and trampolining.

Arc injection exploits also involve at least two primitive operations. The basic arc injection exploit is known as *return-into-libc* because it involved transferring control to C standard library routine **system,** which executes its argument as a new process. The actual command line to execute is provided by the attacker in another primitive operation. [Wojt] describes the "chaining" of multiple return-into-libc exploits for a single defect.

Pure data exploits, by contrast, do not involve any modification of the program's control flow graph. For integer overflows and related defects, this may require only a single primitive operation. For buffer overruns or double-frees, these typically involve a sequence of modifications of addresses used as lvalues or pointers, culminating in the modification of a data value used in some security-critical context.

The most exotic published exploits involve multiple defects. The few existing examples involve taking advantage of an initial defect to gain information about the program in order to enable or improve an exploit of a subsequent defect. [Phen] is most explicit about this, describing how an "information leak" defect is used to make a heap buffer overrun more "stable" (i.e. more consistent). As more powerful mitigation mechanisms are deployed, these *multi-defect exploits* are likely to be a growth area – since most mitigation techniques published to date ignore the possibility of multiple defects.

[Litc] gives an excellent example of a complex real-world exploit for the MS03-026 vulnerability using multiple techniques. **Figure 2** presents a simplified version of the code of the vulnerable program. The **stack_canary** and **global_magic_number** variables are an explicit (albeit simplified) representation of the stack canary mitigation mechanism discussed in more detail below. For the purposes of this section, it is enough to note that the goal of these fragments is to prevent execution from reaching program point P if a buffer overrun has modified the saved return address, which on an x86 machine is located on the stack directly after the stack_canary variable.

By taking advantage of the buffer overrun at P1, Litchfield's exploit first performs the following primitive operations:

- stores a sequence of instructions in the local variable **buff**

- modifies the pointer variable **p** to point to the global variable **global_magic_number**
- sets **i** to a new value **val**
- sets **stack_canary** to a new value **val**
- modifies the saved return address to contain the address of a trampoline that will indirectly transfer execution to **buff**

When execution reaches P2, the assignment modifies the value in the global variable **global_magic_number**, setting it to **val**. As a result, the test at P3 fails to detect the buffer overrun. Execution thus reaches P, where the return instruction results in a control-flow transfer to the saved return value; the trampoline in turn transfers control to the instructions that have been stored in **buff**, and the exploit is successful.

3 Mitigations

The goal of a *mitigation technique* is to reduce an elevation of privilege attack to a denial of service: preferably an orderly termination of the program, but at worst a crash.

A fundamental distinction in mitigation techniques is between those focused on *defects* and those focused on *exploits*. Techniques such as run-time bounds checking that focus on defects have a major advantage: to the extent they are effective, they prevent *all* exploits of those defects. Unfortunately, almost all of these techniques currently have sufficiently-high performance and application compatibility costs that they are not broadly deployed for C/C++ code. By contrast, many mitigation techniques focused on preventing exploits have succeeded in reducing costs to the point where they are broadly deployable. However, exploit-focused mitigations do not prevent other exploits of the same defects, and as a result may not actually provide any protection in practice if it is easy for attackers to switch to other exploits.

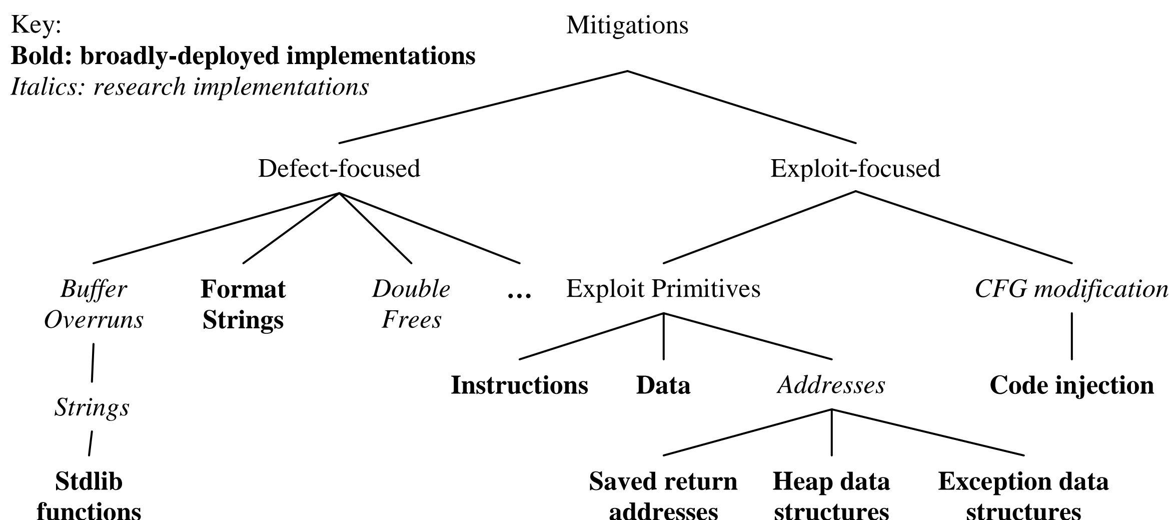
Defect-focused mitigation techniques can be further classified in terms of the defects they protect against. In some cases, a technique protects against only a subset of defects of a given type. For example, [Ruwa] explores focusing explicitly on string buffer overruns (ignoring other buffer overrun possibilities) as a way of reducing overhead.

Exploit-focused mitigation techniques can further be divided into those that attempt to interfere with primitive operations (modification of instructions, data values, and addresses), and those which focus on the higher-level techniques of CFG modification. Some techniques focus only on subsets.

Figure 3 presents this taxonomy of mitigation techniques graphically, highlighting techniques that currently have implementations.

Mitigations operate via one of two mechanisms: *preventing* the defect or exploit from occurring in the first place, or *detecting* the defect or exploit before any damage has occurred. For example, protecting the saved return address prevents "stack smashing" exploits from modifying the return value. Saving an additional copy of the return address and verifying it before returning detects the exploit's attempt to modify it.

Figure 3: Taxonomy of mitigation mechanisms



It is important to note that mitigation techniques potentially have non-trivial costs. Three significant potential costs include run-time performance, application compatibility (for example, [Moln] describes Linux programs that are incompatible with non-executable stack), and the effect on testing and supportability. As a result, most implementations of mitigation mechanisms provide some configuration option to disable them for particular programs or libraries.

Implementations of mitigation techniques may also have limitations (e.g., only covering a subset of possible defects) or vulnerabilities in their own right, and attackers may be able to defeat mitigations exploits by taking advantage of additional vulnerabilities. For example, a randomization technique to prevent address modification will be ineffective if the attacker has a way to obtain the actual (randomized) layout of the program under attack – perhaps through an information leak or incomplete implementation (i.e. only the load addresses are randomized but stack base and heap base are not).

3.1 Defect focused techniques

Many defect-focused techniques attack buffer overruns. *Bounds checking* ensures that any array (or, in C/C++, pointer) access to memory region is between the region's lower and upper bounds. Bounds checking detects buffer over- and under-runs. Languages other than C/C++ (e.g., Pascal, Java, C#, and Ada) often include bounds checking as a feature of the language. The C/C++ language definition not only excludes this, but has several constructs in it that make efficient implementation of bounds checking extremely difficult. [Ruwa] notes that the best-known implementation (the Jones-Kelly **gcc** extensions described in [Jone]) fails on over 50% of programs – that is, programs that do not contain buffer overrun defects terminate abnormally. Performance overhead for bounds checking is very high, typically on the order of 100% or more.

As a result, general-purpose bounds-checking is currently rarely used in practice for C/C++ code. Several current research projects focus on this problem. The CCured system described in [Necu] combines static analysis with run-time checks, but requires source code and compiler modification. [Ruwa] explores modifications of the Jones-Kelly enhancements that succeed on all tested programs and bring overhead of checks down to 130%., and also investigates restricting checking only on string buffers in order to further improve performance.

A substantially more limited, but also lower-overhead, approach to bounds checking is taken by libraries such as LibSafe which provide additional checking for specific "error-prone" C standard library functions. However, these implementations are inherently unable to protect against buffer overruns that do not involve any calls to such functions, and in many cases have other significant limitations as well.

Several variations of defect detection focused on format string vulnerabilities are described in [Cowa01]. The approach implemented in Cowan's FormatGuard is to use C preprocessor macros to remap printf et. al. invocations at compile time to a variant implementation with an additional parameter of the number of format specifiers required (based on the number of arguments provided), and then check the actual number of specifiers in the string provided at run-time. [Cowa01] also notes situations in which this check is insufficient, including a real-world example of a vulnerable program that provided its own printf implementation.

Debugging tools such as Rational's Purify or Compuware's BoundsChecker implement defect detection for uninitialized memory and double-frees of dynamic memory. However, the typical use model of these tools means that the overhead of the checks (frequently up to an order of magnitude slowdown) is typically too large to be deployable on live systems.

A general issue with these techniques is that by revealing previously-latent (but unexploitable) defects, they may have an unfortunate effect on application compatibility. An extreme case of this is discussed in [Cowa01]: while the technique of rejecting any format string with a %n specifier would indeed prove effective at detecting many format string defects, it would also prevent valid programs from running.

3.2 Instructions

An obvious – and very successful – example of a prevention technique is marking the program's code (often known as the text segment) as read-only, as implemented by most operating systems today. This means that any direct attempt to modify the program's instructions will fail.

The "W^X" mitigation technique implemented in OpenBSD generalizes this to ensure that any executable code is marked is read-only. This covers data that is used as a jump table (thus defeating the [Hogl] exploit discussed above), and with the appropriate program modifications can also protect self-modifying programs. Note, however, that a more complex exploit that explicitly includes modifications of memory protection can defeat W^X

as well as the straightforward read-only-text mitigation.

3.3 Data values

A "stack smashing" exploit of a buffer overrun defect is able to modify data values of any local variables laid out above (that is, at a higher memory location than) the buffer being overrun. Laying out scalar and pointer variables in any given stack frame below variables that are likely to be used as buffers (arrays or structs containing arrays) thus protects the non-buffer variables in that frame against being overrun in such stack smashing exploits. ProPolice and Microsoft's Visual C++ both implement variants of this approach.

Alternatively, using a separate stack for variables likely to be used as buffers protects scalar and pointer variables in other stack frames as well, and protects against exploits of buffer underruns.

3.4 Addresses

Two interesting mitigation techniques attempt to to interfering with all address modifications, thus targeting broad classes of exploits. Simpler techniques instead attempt to interfere with modification of specific addresses frequently used in exploits (for example the saved return value, used in stack smashing exploits).

Address space layout randomization (ASLR), originally implemented in the PaX Linux kernel patch, attempts to make the location of global and static variables (as well as program text), unpredictable. To the extent that the randomized layout is undiscoverable and unpredictable, this has the effect of interfering with the attacker's control over any pointer; even if the attacker can put an arbitrary value into the pointer location, she cannot know what it is pointing to. This technique thus defeats CFG modification exploits, and also multi-stage exploits involving pointers.

One potential issue with this approach is the potential for catastrophic failure: if an attacker is able to determine the layout for a particular process (either by taking advantage of an additional defect that leads to an information disclosure, as described in [Bulb], or perhaps by making use of some support or debugging facilities or APIs), the randomization provides no effective protection. In practice, there may also be implementation weaknesses in which certain layout that is not randomized provides a back door to allow the exploit to avoid randomization – for example, an early version of PaX did not randomize the location of the Global Object Table, allowing some straightforward attacks. In addition, "partial overwrites" (where only certain bytes of a pointer are modified) can also defeat randomization in some circumstances.

[Cowa03b] describes *pointer encryption*, a technique to protect arbitrary pointers by encrypting pointer values while in memory and decrypting them before dereferencing. Assuming the encryption cannot be defeated, attackers are thus prevented from controlling a pointer value. As a result, this technique defeats not only CFG modification exploits, but also all multi-stage exploits involving pointers. As of early 2004, no implementation of the technique has been released, and so there has not been any detailed analysis of its limitations; as with randomization, it appears vulnerable to multi-defect exploits.

3.4.1 Saved return addresses

Because many popular exploit techniques involve modification of a saved return address, several mitigation techniques specifically focus on protecting them. Several authors propose storing return addresses on a separate stack, thus preventing them from being modified by stack smashing exploits. [Dahn] propose moving stack buffers to the heap, which similarly prevents modification of return addresses (but in practice provides absolutely no protection, since attackers can always substitute a heap smashing exploit instead). An alterative approach (implemented in the **libverify** library) instead focuses on detecting modification of return addresses by saving a copy at function start and checking it at function return time.

The best-known example of detecting saved return address modifications is the "stack canary" technique, first introduced in [Cowa98] and implemented in combination with other mitigations in various products including StackGuard, ProPolice, and the Microsoft Visual C++ .NET compiler. Essentially, this approach modifies each function to put a distinguishable "canary" on the stack directly before the return value. Before the function returns, the canary is checked, and if its value has been modified, the program aborts. Substantial performance tuning and optimizations reduce the overhead to an acceptable level. Microsoft's Windows Server 2003, various versions of Linux, and OpenBSD are now all built with some variant of stack canary checking.

These approaches are extremely effective at defeating the popular "stack smashing" exploit, and more generally any exploit that directly uses a buffer overrun to modify the return address. When other exploit techniques are possible for a defect, however, stack canaries by themselves can thus be easily defeated; see, for example, [Litc].

3.4.2 Heap data structures

The technique of heap meta-data checking specifically focuses on protecting pointers used in the implementation of dynamic memory allocators. Essentially, this technique performs additional checks of private data structures to ensure that the intended invariants have not been compromised by some unexpected modification of memory. This is an extremely effective mitigation against all published exploits for double-free defects, as well as the general "heap smashing" exploits for heap buffer overruns. However, an insufficient implementation may by bypassable (for example, [Phen] discusses a successful search for un-validated meta-data when exploiting a Cisco IOS heap overrun).

3.4.3 Exception data structures

The SafeSEH mechanisms introduced in Microsoft VC++ .NET are another example of detecting modification of specific pointers, in this case those contained in exception handling structures. This additional checking defeats the "exception handling clobbering" attack.

3.5 CFG modification

Much work on mitigations has gone into attempts to defeat code injection, or more generally CFG modification exploits. This is not surprising, since most published exploits, and all high-profile worms to date, use these techniques. However, it is important to note that such mitigations provide no protection against other forms of exploits, and thus despite many authors' claims these mitigations by themselves can not fully prevent exploitation of buffer overruns or other defects.

Making certain areas of memory non-executable, a collection of techniques collectively referred to as *NX,* prevents code-injection (but not arc-injection or pure data) exploits. A non-executable stack defeats the standard stack smashing exploit (and more generally, any code injection exploit where the code is injected onto the stack): the attempt to execute instructions the attacker has stored on the stack fails, leading to program termination and effective mitigation. Non-executable heaps and the so-called W^X generalization (no page in memory can be simultaneously writeable and executable) extend this to other areas of memory.

Some existing programs do intentionally create and execute instructions on the heap and stack (e.g., JIT compilers, some graphics drivers, and certain 'thunking' layers), so there are typically some application compatibility issues; these typically are handled by disabling the mitigation for specific applications.

These are extremely popular mitigation strategies, due to their effectiveness at preventing the "easiest" exploits and the belief that they combine well with other techniques. Many chips provide hardware support for marking pages or segments non-executable, which makes this an attractive option from a performance perspective; Solaris and 64-bit versions of Microsoft Windows Server 2003 provide support for NX; OpenBSD 3.3 supports W^X; patches and third-party products also provide this support for Linux and Windows XP; and Microsoft has announced NX support for the upcoming Windows XP SP2 on CPUs that provide appropriate levels of hardware support.

Systrace is a mechanism for constraining an application's access to the system by restricting what system calls the application can make. Depending on the behavior of the program being protected by systrace, this allows detection of some common code injection exploits; however, it offers no protection against arc injection or pure data exploits. Systrace is commonly viewed as a "second line" of defense that attempts to limit the damage performed by an exploit that has succeeded in elevation of privilege.

Program shepherding [Kiri] is a much more ambitious approach that attempts to prevent all unexpected control flow transfers – that is, all CFG modification attacks, specifically including arc injection as well as code injection. While this prevents a broader class of exploits, the performance overhead is potentially more severe, and there are significant complexities related to exception handling, function-pointer variables, and self-modifying code. Currently, this is still a research technique.

4 Incomparability of mitigations

Since each mitigation technique adds cost and complexity to the system, it is obviously desirable to minimize the number of techniques that are required. Unfortunately, as we show in this section, all pairs of well-analyzed existing mitigations are *incomparable*: technique A prevents some exploits for some defects that are not prevented by mitigation B, but mitigation B prevents some exploits for some defects that are not prevented by mitigation A.[1]

[1] [The lack of analysis to date of attacks on program shepherding leaves open the possibility that it may be strictly stronger than NX.]

Table 4: Incomparable mitigation techniques

Family	Classification	Example techniques
Defect focused techniques	Defect	bounds checking, format string checking, garbage collection
Non-writable code segment	Exploit: instruction	
local variable protection	Exploit: data values	local variable reordering or separate stack for buffers
return address protection	Exploit: saved return addresses	stack canaries, separate return value stack
heap meta-data checking	Exploit: heap data structures	
general pointer protection	Exploit: addresses	address space randomization, pointer encryption
NX (non-executable regions of memory)	Exploit: code injection	
program shepherding	Exploit: CFG modification	

This result is a consequence of the classification of mitigation techniques discussed in Section 4. We first provide a high-level sketch of our approach and then a more detailed (although still informal) description.

There are three interesting sub-cases that cover most pairs of mitigation techniques:

- Defect-focused mitigations are incomparable with exploit-focused mitigations.
- Exploit-focused mitigations that defeat one exploit primitive are incomparable with exploit-focused mitigations that defeat other exploit primitives.
- Mitigations that defeat code- and arc-injection are incomparable with mitigations that defeat data and instruction modification.

Case-by-case analysis is then required for the remaining pairs, primarily the multiple families of mitigations that defeat various address modification exploits.

4.1 Defect-focused vs. exploit-focused mitigations

Defect-focused and exploit-focused mitigation techniques are fundamentally incomparable. A simple example makes the point intuitively, using bounds checking as an example of a defect-focused mitigation technique and NX as an example of an exploit-focused technique. NX not only prevents code injection exploits for buffer overrun defects (as does bounds checking), but also for other kinds of defects such as format string attacks that are not addressed by bounds checking. Conversely, bounds checking prevents other categories of exploits for buffer overruns that are not addressed by NX. More generally, bounds checking prevents pure data exploits of buffer overruns – exploits that are not prevented by any existing exploit-focused mitigation technique.

A similar argument can be made for any combination of a defect-focused and an exploit-focused mitigation assuming that there are multiple possible exploit techniques for at least some defects prevented by the defect-focused mitigation, and the exploit-focused mitigation applies to at least some defects of multiple types. These assumptions hold with respect to the exploit and mitigation techniques discussed in this paper.

As this example shows, the possibility of currently-unknown exploit techniques tends to be an argument for defect-focused mitigations. On the other hand, there is also a possibility of currently-unknown defects that are exploitable by existing exploit techniques; this is an argument in favor the complementary value of exploit-focused mitigations.

4.2 Exploit-focused mitigations

The incomparability of mitigations that defeat different exploit primitives is trivial: the sets of single-operation exploits (that is, exploits in which the attacker only uses a single primitive operation) they prevent are by definition disjoint. The incomparability of mitigations focused on code- and arc-injection with mitigations focused on data value or instruction modifications is also immediately obvious.

We now must proceed to case-by-case analysis of the remaining pairs of mitigations.

Heap meta-data checking is intuitively incomparable with return address protection because they protect disjoint memory locations. Specifically, heap meta-data checking protects against exploits of heap buffer overruns and double frees, but not against

stack buffer overruns; return address protection is the exact opposite.

All published general pointer protection mitigations are vulnerable to exploits involving an earlier information disclosure. Conversely, however, they protect against exploits of all kinds of addresses (unlike other address protection mitigations) and against multi-stage pure data exploits (unlike mitigations that prevent code- and arc-injection).

Program shepherding defeats arc-injection exploits, unlike NX. Little work has been done analyzing weaknesses of program shepherding, and so it is not clear whether the converse also holds. The prototype implementation of program shepherding has several known vulnerabilities (limitations on handling of indirect calls, an internal data structure that is not protected) that are likely to allow exploits that would be prevented by NX, but [Kiri] has sketched extensions that may address this. It is possible that with sufficiently precise analysis of indirect jumps (including handling of C++ virtual functions and exceptions, C longjmps, and re-analysis after a program's control-flow graph is modified at runtime via loading and unloading of dynamic link libraries) program shepherding may prove to be strictly stronger than NX. These solutions have not yet been implemented in practice, so currently program shepherding is indeed incomparable with NX, but this may change in the future.

4.3 Implementation-based considerations

The discussion above concentrated on the inherent limitations of the various mitigation techniques. In practice, the actual implementation of techniques is also a potentially-important source of incomparability.

One important instance of this relates to the difference between mitigations that require recompilation and those that do not. Currently, implementations of stack canary and bounds checking schemes require recompilation, and so do not provide any protection to software that is not recompiled. Some implementations of techniques such as NX and heap meta-data checking, by contrast, apply at the system wide level, and so have the potential to protect most un-recompiled applications as well – although as noted above, some applications involving on-the-fly code generation require modification for NX compatibility.

Any implementation may well have limitations. For example, many stack canary implementations have optimizations that disable checking on "small" objects; it is certainly possible to construct situations in which exploitable buffer overruns occur in those small objects and the exploit is not detected due to optimizations. Application compatibility concerns may also lead to weaknesses; NX schemes, for example, typically have to provide some backward compatibility switch for applications that generate instructions on the fly, and so those applications may receive less (or no) protection.

Finally, implementations may have weaknesses or flat-out bugs. Early versions of some stack canary checks were vulnerable to exploits where the attacker could guess the canary. [Litc] discusses implementation weaknesses, since fixed, in Microsoft's SafeSEH implementation; and [Bulb] presents attacks that took advantage of incomplete randomizations in early versions of PaX.

5 Combinations of mitigations

The most powerful currently-deployed approaches to mitigation combine multiple complementary techniques. This is a natural response to the incomparability of mitigations, and some combinations are obviously attractive – for example, stack canaries apply only to stack buffer overruns of stack memory, and heap meta-data checking applies only to heap buffer overruns, so the two are clearly complementary.

More generally, the possibility of implementation weaknesses is another argument in favor of combining mitigations. Of course, the additional costs (including system complexity) of using multiple mitigation techniques need to be balanced against the benefits.

Assuming that the techniques being combined are independent (in the sense that they have no interactions), a combination of multiple techniques mitigates an individual vulnerability if any one technique mitigates it.

Unfortunately, currently-deployed combinations of mitigation techniques – and more generally, any combination of currently published mitigations – do not provide even the weaker level of protection of preventing all published exploits. It is important to note that this is not simply a matter of implementation weaknesses. Pure data exploits of integer overflows and signed/unsigned mismatches are simply not addressed by any published mitigation technique. Pure data exploits of buffer overruns are only addressed by the defect-oriented technique of bounds checking, which is not yet widely deployed.

Table 5: Combinations of mitigations and some known unmitigated exploits (ignoring implementation weaknesses)

Techniques	Some known unmitigated exploit techniques
NX + return value protection + heap meta-data checking	Pure data; some arc injection
return value protection + heap meta-data checking	Pure data (direct and multi-stage); some multi-stage arc/code injection
return value protection + heap meta-data checking + variable reordering + NX	Most pure data; some arc injection
NX + pointer protection	Most pure data; arc injection via partial overwrites; using previous information disclosures vulnerabilities
bounds checking + heap meta-data checking + format string checking	Some pure data (e.g., for integer overflows, signed/unsigned mismatches, etc.)
NX + pointer protection + format string checking + variable reordering	Some pure data (e.g., for some buffer overruns, integer overflows, signed/unsigned mismatches, etc.); using previous information disclosures vulnerabilities

6 Future work

Exploits for race conditions are often viewed as being a "next wave" of vulnerabilities; conceptually, these (and the mitigations) should fit cleanly into this analysis framework. Moving beyond low-level defects to such higher-level (and non-C/C++) constructs as script injection and SQL injection, may require consideration of new exploit techniques; however, it is interesting to note that this is essentially a form of instruction modification, just at a higher level of "instruction".

Better understanding the impact of actual implementations of mitigation techniques is likely to require more precise analysis of the specific characteristics of exploits and mitigations. One potential approach is to extend the Dor et. al. memory model (originally targeted at static analysis of buffer overrun defects) to the full complexity needed to consider exploits and mitigations as well.

More fundamentally, an important question not addressed in this paper is what percentage of vulnerabilities is exploitable for elevation of privilege in the presence or absence of specific mitigations. With respect to known exploit techniques, this requires estimating what percentage of defects is amenable to various exploit techniques, and the effectiveness of various mitigation techniques at defeating the exploits. This quantification will allow better understanding of the actual strength of a system in the presence of combinations of mitigations – in light of the a well-known truism that "there is no such thing as perfect security", do some of the combinations of mitigations discussed above give sufficient protection in practice that exploitation of low-level defects will cease to be a significant elevation of privilege threat? Combining this information with cost estimates will also enable more meaningful cost/benefit tradeoffs between mitigation techniques.

The specific mitigations discussed in this paper can be viewed as special cases of more general "defenses" that include firewalls, intrusion detection systems, program sandboxing, etc. Mitigations interfere with exploits by preventing some of the "environmental conditions" needed for a successful exploit; so do the general defenses. Combining this work with cost/benefit analyses such as in [Butl] is a longer-term goal.

7 Related work

Exploits first received significant attention in response to the "Internet worm" (also know as the *fingerd* worm); [Spaf] is the definitive writeup. Subsequent work has rarely been published in traditional journals but is often of very high quality; [Litc] is a good example. Cowan's work, especially [Cowa00, Cowa01], and [Wila] reference many of the key papers from *Phrack* and elsewhere. [Hogl] is a recent book-length publication focused on exploits and contains significant references. None of these works reduce exploit techniques to their primitives.

To date, there have been few significant efforts at collecting, let alone classifiying, mitigations. [Cowa00] discusses a handful of techniques. [Wila] provides a simple partial taxonomy of several mitigations focused on CFG-modification; more importantly, this work compares the effectiveness of several mitigation techniques against a testbed of roughly 20 "attack patterns" (exploit techniques, in

our terminology), and finds that none are particularly effective – all miss at least 50% of the attacks. As with exploits, there is substantial work on individual mitigation techniques. Crispin Cowan's StackGuard [Cowa98] and SolarDesigner's non-executable stack Linux kernel patch (described in [Wojt]) were amongst the first high-profile mitigation approaches.

Research continues on more ambitious mitigation techniques. Reducing overhead to an acceptable level is a major focus defect-focused mitigations concentrating on buffer overruns; [Ruwa] and [Necu] are tood examples of this. Program shepherding [Kiri] and pointer encryption [Cowa03b], by contrast, explore more powerful exploit-focused mitigations.

Static analysis of defects is a related field; obviously, removing defects is even better than mitigating the resulting vulnerability; [Cowa03a] includes discussion of static analysis tools. [Xie], [Dor] and others are good examples of more recent work. Unfortunately, none of these tools find all the defects in real cases; for example, while many of the tools have found both previously-reported and (in some cases) new buffer overrun defects in Sendmail, additional defects have since been found via manual inspection. Recent work such as CCured [Necu] combines static and dynamic techniques, but requires modification to the source code, and has not yet been broadly deployed.

8 References

[Bulb] Bulba and Kil3r, "Bypassing Stackguard and Stackshield", *Phrack* 56, 2000.

[CERT] "Linux kernel do_brk() function contains integer overflow", *CERT/CC Vulnerability Note VU#301156*, December, 2003.

[Cowa98] C. Cowan et. al., *StackGuard: Automatic Adaptive Detection and Prevention of Buffer overrun attacks*, 1998.

[Cowa00] C. Cowan et. al., "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade". In *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000.

[Cowa01] C. Cowan et. al., "FormatGuard: Automatic from printf Format String Vulnerabilties". In *Proceedings of the 2001 USENIX Security Symposium*, August 2001.

[Cowa03a] C. Cowan, "Software Security for Open Source Systems",*IEEE Security & Privacy Magazine*, February 2003.

[Cowa03b] C. Cowan et. al., "PointGuard: Protecting Pointers from Buffer Overrun Vulnerabilties". In *Proceedings of the 2003 USENIX Security* Symposium, August 2003.

[Dahn] C. Dahn and S. Mancoridis, "Using Program Transformation to Secure C Programs Against Buffer Overflows", In IEEE *Proceedings of the 2003 Working Conference in Reverse Engineering (WCRE'03)*, November 2003.

[Dor] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overruns in C", in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (PLDI03), June 2003.

[Hogl] G. Hoglund and G. McGraw *Exploiting Software: How to Break Code*, Addison Wesley, 2004.

[Kiri] V. Kiriansky et. al., "Secure execution via program shepherding", In *Proceedings of the 2002 USENIX Security Symposium*, August 2002.

[Litc] 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

[Moln] I. Molnar, *"Exec Shield", new Linux security feature*, 2003. http://lists.insecure.org/lists/linux-kernel/2003/May/0371.html

[Necu] G. Necula et. al., "CCured in the Real World," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (PLDI03), June 2003.

[Phen] Phenolit, *Ultima Ratio: A Remote Cisco IOS Exploit*, 2002. http://www.phenoelit.de/ultimaratio/index.html.

[Ruwa] O. Ruwase and M. S. Lam, "A Practical Dynamic Buffer Overflow Detector", In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.

[Spaf] E. Spafford. *The Internet Worm Program: An Analysis* . Technical Report CSD-TR-823. Department of Computer Science, Purdue University. November 1988.

[Xie] Y. Xie and D. Engler, "Using Redundancies to Find Errors", to appear in IEEE *Transactions on Software Engineering.*

[Wila] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention", in *Proceedings of the 10th Network and Distributed System Security Symposium*, November 2003.

[Wojt] R. Wojtczuk, *Defeating Solar Designer's Non-executable Stack Patch*, 1998. http://www.insecure.org/sploits/non-executable.stack.problems.html