Runtime countermeasures for code injection attacks against C and C++ programs

YVES YOUNAN, WOUTER JOOSEN and FRANK PIESSENS
Katholieke Universiteit Leuven

The lack of memory-safety in C/C++ often leads to vulnerabilities. *Code injection attacks* exploit these to gain control over the execution-flow of applications. These attacks have played a key role in many major security incidents. Consequently, a huge body of research on countermeasures exists. We provide a comprehensive and structured survey of vulnerabilities and countermeasures that operate at runtime. These countermeasures make different trade-offs in terms of performance, effectivity, compatibility, etc. This makes it hard to evaluate and compare countermeasures in a given context. We define a classification and evaluation framework, on the basis of which countermeasures can be assessed.

Categories and Subject Descriptors: D.3.0 [**Programming Languages**]: General—*Vulnerabilities*; *Protection Mechanisms*

General Terms: Security

Additional Key Words and Phrases: Code injection, countermeasures, C, C++

1. INTRODUCTION

Software vulnerabilities have been a major cause of computer security incidents since the advent of multiuser and networked computing [Microsoft 2003; Spafford 1989]. Most of these software vulnerabilities can be traced back to a few mistakes that programmers make over and over again. Even though many papers and books [Howard and LeBlanc 2001; Viega and McGraw 2002] attempt to teach programmers how to program more securely, the problem persists and will most likely continue to be a major problem in the foreseeable future. This paper focuses on a specific subclass of software vulnerabilities, implementation errors in C and C++, as well as the countermeasures that have been proposed and developed to deal with these vulnerabilities. More specifically, implementation errors that allow an attacker to break memory safety and execute foreign code are addressed in this survey.

Several preventive and defensive countermeasures have been proposed to combat exploitation of common implementation errors, and this paper examines many of these. We also describe several ways in which some of the proposed countermeasures can be circumvented. The paper focuses on *run-time* countermeasures: only countermeasures that have some effect at run-time are in scope. This includes countermeasures that perform additional run-time checks, or harden the C/C++ run-time environment. It excludes purely static countermeasures, for instance those that try to detect vulnerabilities using static analysis or program verification. It also excludes testing approaches such as fuzzers. These areas are also very active research areas and deserve their own survey. Preliminary attempts at such a survey can be found in [Wilander and Kamkar 2002; Pozza et al. 2006].

Although some countermeasures examined here protect against the more general case of buffer overflows, this paper focuses on protection against attacks that specifically attempt to execute code an application would not execute in normal circumstances. Such attacks subvert the control flow of the application either to injected code or to existing code which is then executed in a different context.

Given the large number of runtime countermeasures that have been proposed to deal with such attacks, and given the wide variety in techniques used in the design of these countermeasures, it is hard for an outsider of the research field itself to get a good understanding of existing solutions. This paper aims to provide such an understanding to software engineers and computer scientists without specific security expertise, by providing a structured classification and evaluation framework. At the top level, we classify existing countermeasures based on the main technique they use to address the problem.

Safe languages are languages in which most of the implementation vulnerabilities do not exist or are hard to exploit. These languages generally require a programmer to specifically implement a program in this language or to port an existing program to this language. We will focus on languages that are similar to C, i.e., languages that stay as close to C and C++ as possible. These are mostly referred to as safe dialects of C. Programs written in these dialects generally have some restrictions in terms of memory management: the programmer no longer has explicit control over the dynamic memory allocator.

Bounds checkers perform bounds checks on array and pointer operations and detect when the program tries to perform an out of bounds operation and take action accordingly.

Probabilistic countermeasures make use of randomness to make exploitation of vulnerabilities harder.

Separators and replicators of information exist in two types: the first type will try to replicate valuable control-flow data or will separate this data from regular data. Replication can be used to verify the original value, while separation prevents an attacker from overwriting the separated data because it is no longer adjacent to the vulnerable object. The second type relies on replication only, but replicates processes with some diversity introduced. If the processes act differently for the same input, then an attack has been detected.

VMM-based countermeasures make use of properties of the virtual memory manager to build countermeasures.

Execution monitors observe specific security-relevant events (like system calls) and perform specific actions based on what is monitored. Some monitors will try to limit the damage of a successful attack on a vulnerability on the underlying system by limiting the actions a program can perform. Others will detect if a program is exhibiting unexpected behavior and will provide alerts if this occurs. The first type of runtime monitor is called a sandbox, while the second type of monitoring is called anomaly detection.

Hardened libraries replace library functions with versions that perform extra checks to ensure that the parameters are correct.

Runtime taint trackers will instrument the program to mark input as tainted. If such tainted data is later used in the program where untainted data is expected or is used to modify a trusted memory location (like a return address), then a fault is generated.

This paper is structured as follows: Section 2 contains an overview of the implementation errors that the countermeasures in Section 4 defend against. It also describes typical ways in which these implementation errors can be abused. Section 3 contains a description of the properties that we will assign to the various countermeasures that are examined in Section 4. Section 4 contains our survey of countermeasures and in some cases, ways in which they can be circumvented. Section 5 presents our conclusion.

2. IMPLEMENTATION VULNERABILITIES AND EXPLOITATION TECHNIQUES

This section contains a short summary of the implementation errors for which we shall examine countermeasures. It is structured as follows: for every vulnerability we first describe why a particular implementation error is a vulnerability. We then describe the basic technique an attacker would use to exploit this vulnerability and then discuss more advanced techniques if appropriate. We mention the more advanced techniques because some of these can be used to circumvent some countermeasures. A more thorough technical examination of the vulnerabilities and exploitation techniques (as well as a technical examination of some countermeasures) can be found in [Younan 2003; 2008; Erlingsson et al. 2010].

When we describe the exploitation techniques in this section, we focus mostly on the IA32-architecture [Intel Corporation 2001] even though other architectures are also vulnerable to these attacks [Francillon and Castelluccia 2008; Younan et al. 2009]. While the details for exploiting specific vulnerabilities are architecture dependent, the main techniques presented here should be applicable to other architectures as well.

2.1 Stack-based buffer overflows

2.1.1 *Vulnerability.* When an array is declared in C, space is reserved for it and the array is manipulated by means of a pointer to the first byte. At runtime no information about the array size is available and most C compilers will generate code that will allow a program to copy data beyond the end of an array, overwriting adjacent memory space. If interesting information is stored somewhere in such adjacent memory space, it could be possible for an attacker to overwrite it. On the stack this is usually the case: it stores the addresses to resume execution after a function call has completed its execution.

On the IA32-architecture the stack grows down (meaning newer stackframes and variables are at lower address than older ones). The stack is divided into stackframes. Each stackframe contains information about the current function: arguments to a function that was called, registers whose values must be stored across function calls, local variables, the saved frame pointer and the return address. An array allocated on the stack will usually be contained in the section of local variables of a stackframe. If a program copies past the end of this array, it will be able to overwrite anything stored before it, and it will be able to overwrite the function's management information, like the return address.

Figure 1 shows an example of a program's stack when executing the function $f1$.

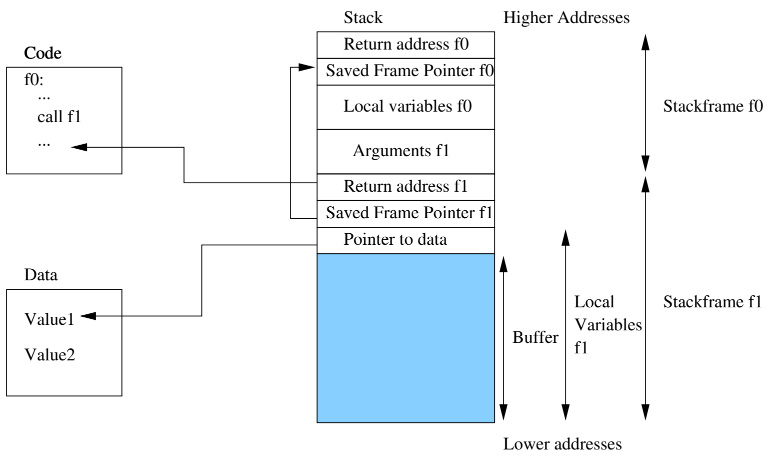

Fig. 1. Stack-layout on the IA32

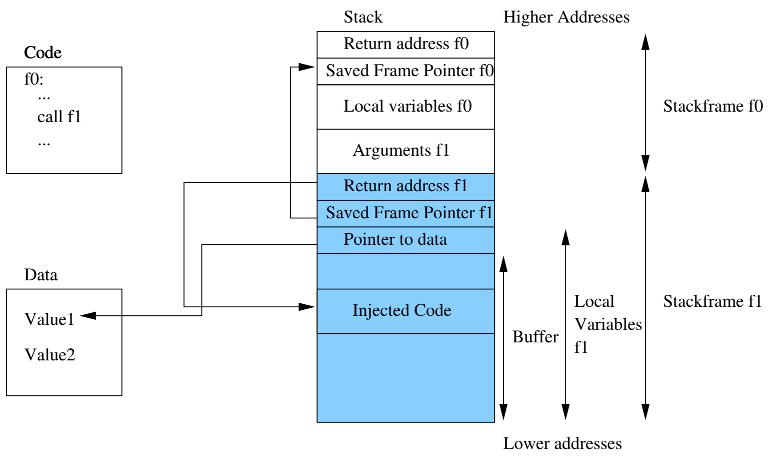

Fig. 2. Stack-based buffer overflow

This function was called by the function $f0$, that has placed the arguments for $f1$ after its local variables and then executed a call instruction. The call has saved the return address (a pointer to the next instruction after the call to $f1$) on the stack. The function prologue (a piece of code that is executed before a function is executed) then saved the old frame pointer on the stack. The value of the stack pointer at that moment has been saved in the frame pointer register. Finally space for two local variables has been allocated: a pointer pointing to data and an array of characters (a buffer). The function would then execute as normal. The colored part indicates what could be written to by the function if the buffer is used correctly.

2.1.2 *Exploitation.*

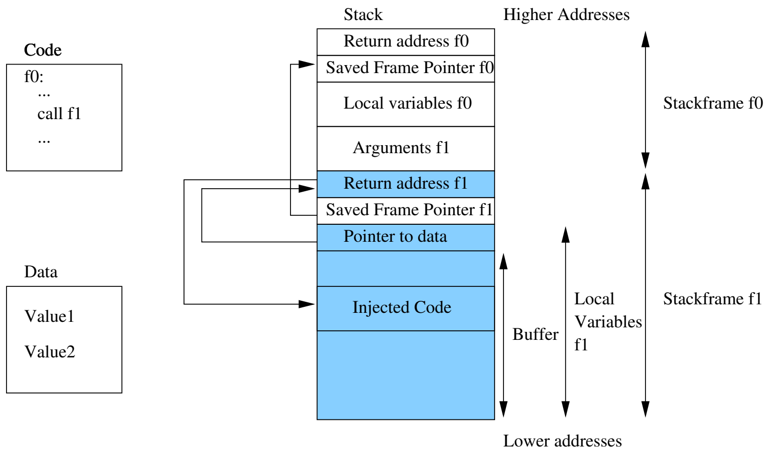

Fig. 3. Stack-based buffer overflow using indirect pointer overwriting

2.1.2.1 *Basic exploitation.* Figure 2 shows what could happen if attackers are able to make the program copy data beyond the end of an array. Besides the contents of the buffer, the attackers have overwritten the pointer, the saved frame pointer (these last two have been left unchanged in this case) and the return address of the function. They could continue to write into the older stackframe if so desired, but in most cases overwriting the return address is an attacker's main objective as it is the easiest way to gain control over the program's execution-flow. The attackers have changed the return address to point to code that they copied into the buffer, probably using the same copying operation that they used to copy past the end of the buffer. When the function returns, the return address would, in normal cases, be used to resume execution after the function has ended. But since the return address of the function has been overwritten with a pointer to the attacker's injected code, execution-flow will be transferred there [Aleph One 1996; Smith 1997].

2.1.2.2 *Indirect pointer overwriting.* If attackers, for some reason, cannot overwrite the return address directly (some countermeasures prevent this), they can use a different technique illustrated in Figure 3 called indirect pointer overwriting [Bulba and Kil3r 2000] which might still allow them to gain control over the execution-flow.

The overflow is used to overwrite the local variable of $f1$ holding the pointer to value1. The pointer is changed to point to the return address instead of pointing to value1. If the pointer is then dereferenced and the value it points to is changed at some point in the function $f1$ to an attacker-specified value, then the attacker can use it to change the return address to a value of their choosing.

Although in our example we illustrate this technique by overwriting the return address, indirect pointer overwriting can be used to overwrite arbitrary memory locations: any pointer to code that will be executed later could be interesting for an attacker to overwrite.

2.2 Heap-based buffer overflows

2.2.1 *Vulnerability.* Heap memory is dynamically allocated at run-time by the application. As is the case with stack-based arrays, arrays on the heap can, in most implementations, be overflowed too. The technique for overflowing is the same except that the heap grows upwards in memory instead of downwards. However no return addresses are stored on the heap, so an attacker must use other techniques to gain control of the execution-flow.

2.2.2 *Exploitation.*

2.2.2.1 *Basic exploitation.* One way of exploiting a buffer overflow located on the heap is by overwriting heap-stored function pointers that are located after the buffer that is being overflowed [Conover 1999]. Function pointers are not always available though, so other methods of exploiting heap-based overflows are by overwriting a heap-allocated object's virtual function pointer [rix 2000] and pointing it to an attacker-generated virtual function table. When the application attempts to execute one of these virtual methods, it will execute the code to which the attacker-controlled pointer refers.

2.2.2.2 *Dynamic memory allocators.* Function pointers or virtual function pointers are not always available when an attacker encounters a heap-based buffer overflow. Overwriting the memory management information that is generally associated with a dynamically-allocated block [Solar Designer 2000; anonymous 2001; Kaempf 2001; BBP 2003] is a more general way of exploiting a heap-based buffer overflow.

Overwriting the memory management information is a specific instantiation of an indirect pointer overwrite: the pointers in the memory management information get modified to point to a code pointer, when the memory manager later attempts to use the memory management information (e.g. to free a chunk), it will overwrite this code pointer. Given that many memory managers store this memory management information in-band (usually right before the data stored in the chunk of memory), if a heap overflow exists, an attacker can often overwrite the management information [Younan et al. 2010].

2.3 Dangling pointer references

2.3.1 *Vulnerability.* A pointer to a memory location could refer to a memory location that has been deallocated either explicitly by the programmer (e.g. by calling free()) or by code generated by the compiler (e.g. a function epilogue, where the stackframe of the function is removed from the stack). Dereferencing of this pointer is generally unchecked in a C compiler, causing the dangling pointer reference to become a problem. In normal cases this would cause the program to crash or exhibit uncontrolled behavior.

However, in some specific cases, if the program continues to write to memory that has been released and reused, it could also result in an attacker being able to overwrite information in a memory region to which he was never supposed to write. Even reading of such memory could result in a vulnerability where information stored in the reused memory is leaked.

A specific example of a such a write-after-free problem, is the double free vulnerability. A double free vulnerability occurs when already freed memory is deallocated a second time. This could again allow an attacker to overwrite arbitrary memory locations [Dobrovitski 2003].

2.3.2 *Exploitation.* If the program reuses memory that was freed earlier for an object, but a dangling pointer remains in the program for an object of a different type, then that dangling pointer could be used to modify the new object. For example, if the program had allocated memory for a buffer that stores user input but it has freed this memory, suppose it is now reused for an object containing a function pointer. If a dangling pointer exists to the buffer, a user may be able to modify the function pointer of the new object, possibly resulting in execution of injected code. However, this type of exploit is very program specific.

It is also possible to exploit the memory allocator in a similar way: in Linux the memory allocator will store free chunks in a doubly linked list of free chunks. This doubly linked list is implemented by storing a forward and a backward pointer in the chunk, over the area where the data was stored when the chunk was in use. Because there is such a difference between a used and an unused chunk, if a dangling pointer exists that points to memory, an attacker could modify the forward and backward pointers. Such a modification can lead to an indirect pointer overwrite when the memory allocator tries to remove this chunk from the doubly linked list.

2.4 Format string vulnerabilities

2.4.1 *Vulnerability.* Format functions are functions that have a variable amount of arguments and expect a format string as argument. This format string will specify how the format function will format its output. The format string is a character string that is literally copied to the output stream unless a % character is encountered. This character is followed by format specifiers that will manipulate the way the output is generated. When a format specifier requires an argument, the format function expects to find this argument on the stack. For example, consider the following call: $printf(''\%d'', d)$. Here printf expects to find the integer d as second argument to the printf call on the stack and will read this memory location and output it to screen. A format string vulnerability occurs if an attacker is able to specify the format string to a format function (e.g. $printf(s)$, where s is a user-supplied string). The attacker is now able to control what the function pops from the stack and can make the program write to arbitrary memory locations. Chen and Wagner [2007] suggest that format string vulnerabilities are more common than previously thought, although not as prevalent as buffer overflows: they found 1533 possibly format string vulnerabilities (of which they assume 85% are real vulnerabilities) on 92 million lines of code that they analyzed on a Debian Linux system.

2.4.2 *Exploitation.* One format specifier is particularly interesting to attackers: $\%n$. This specifier will write the amount of characters that have been formatted so far to a pointer that is provided as an argument to the format function [ISO 1999].

Thus if attackers are able to specify the format string, they can use format specifiers like $\%x$ (print the hex value of an integer) to pop words off the stack, until they reach a pointer to a value they wish to overwrite. This value can then be overwritten by crafting a special format string with $\%n$ specifiers [scut 2001]. However addresses are usually large numbers, especially if an attacker is trying to execute code from the stack, and specifying such a large string would probably not be possible. To get around this a number of things must be done. Firstly format functions also accept minimum field width specifiers when reading format specifiers. The amount of bytes specified by this minimum field width will be taken into account when the $\%n$ specifier is used (e.g. $printf(''\%08x'', d)$ will print d as an eight digit hexadecimal number: if d has the decimal value 10 it would be printed as $0000000a$). This field width specifier makes it easier to specify a large format string but the number attackers are required to generate will still be too large to be used effectively. To circumvent this limitation, they can write the value in four times: overwriting the return address with a small value (normal integers on the IA32 will overwrite four bytes), then overwriting the return address + one byte with another integer, then return address + two bytes and finally return address + three bytes.

The attacker faces one last problem: the amount of characters that have been formatted so far is not reset when a $\%n$ specifier is written. If the address the attackers want to write contains a number smaller than the current value of the $\%n$ specifier, this could cause problems. But since the attackers are writing one byte at a time using a four byte value, they can write larger values with the same least significant byte (e.g. if attackers want to write the value $0x20$, they could just as well write $0x120$).

2.5 Advanced exploitation techniques

While the previous sections discussed basic exploitation techniques for the types of vulnerabilities discussed, due to the emergence of countermeasures in commodity operating systems, attackers have developed more advanced exploitation techniques that are able to bypass some of the protections. This section gives a short overview of some of the more important of these advanced techniques: these techniques are generally applicable for exploitation of all the previously mentioned vulnerabilities, either in combination with the basic exploitation technique or without.

Return-to-libc attacks [Wojtczuk 1998], or return oriented programming in the more general sense [Shacham 2007], are a result of operating systems enforcing non-executable permissions for data sections in memory, meaning that the attackers can no longer inject and execute their code. These attacks make use of existing code in the program to execute an attack. If the attacker passes different information to the existing code (either through the stack or by setting different values in registers), the meaning of the original instructions can be changed. Everything that is present in the code section of the program can be abused by an attacker: if it is possible, as is the case in Intel architectures, to jump into the middle of instructions, then the processor may interpret certain instructions completely differently. Attackers will then generally look for a pattern of interesting instructions followed by a return instruction because this returns control to the attackers if they have control of the stack. Repeated application of this pattern allows the attacker to execute arbitrary code [Shacham 2007].

Heap-spraying attacks [SkyLined 2004] came into existence because they allow an attacker to more reliably exploit a program even if it's harder for the attacker to figure out where his injected code is in the program's address space. For example, this could be the result of the use of a countermeasure that randomizes the address space, or because it is hard for the attacker to predict the heap layout because of different program usage in different situations. The attack will generally make use of a scripting language available in the program (e.g. javascript) or some other means that allows the attacker to import large amounts of data into the program's memory (e.g. images, movies, etc.). The most used instantiation of this attack is against browsers: it uses javascript to fill the browser's memory with injected code (e.g. 1GB of memory). Attackers can then predict that at one of the higher memory regions there is a high probability that their injected code is located and can then use one of the previously mentioned vulnerabilities to direct control flow to this location.

Non-control data attacks [Chen et al. 2005] overwrite information that is security sensitive in the program but do not necessarily subvert the program's control flow. Such an attack could for example overwrite a user id in a program, potentially giving the attacker increased privileges. While these types of attacks are out of scope for this paper since we focus on control flow subversion, it is worth remembering that these types of attacks could be used by attackers to bypass some of the countermeasures discussed in Section 4.

3. COUNTERMEASURE PROPERTIES

This paper aims to provide an understanding of the field of code injection attacks to software engineers and computer scientists without specific security expertise. We do this by providing a structured classification and evaluation framework of countermeasures that exist to deal with these types of attacks. At the top level, we classify existing countermeasures based on the main technique they use to address the problem.

However, countermeasures also make different trade-offs in terms of performance, effectivity, memory cost, compatibility, etc. In this section we define a number of properties that can be assigned to each countermeasure. Based on these properties, advantages and disadvantages of different countermeasures can be assessed.

3.1 Type

The types of protection that countermeasures provide are contained in Table I. Countermeasures that offer detection will detect an attack when it occurs and take action to defend the application against it, but will not prevent the vulnerability from occurring. Prevention countermeasures attempt to prevent the vulnerability from existing in the first place and as such are generally not able to detect when an attacker is attempting to exploit a program as the vulnerability should have been eliminated. Countermeasures that make it harder for an attacker to exploit a vulnerability but that do not actually detect an attempt as such are of the type mitigation. Finally the last type of countermeasures do not try to detect, prevent or mitigate an attack or a vulnerability but try to contain the damage that an attacker can do after exploiting a vulnerability.

3.2 Vulnerabilities

Table II contains a list of the vulnerabilities that the countermeasures in this paper address. They reflect the scope of the vulnerabilities that the designer of the countermeasure wished to address.

Table I. Type

Code	Type
D	Detection: exploitation attempts are detected
P	Prevention: The vulnerability is prevented
M	Mitigation: Exploitation is made harder, no explicit detection
C	Containment: Limits the damage of exploitation

Table II. Vulnerability

Code	Vulnerability
S	Stack-based buffer overflow
H	Heap-based buffer overflow
D	Dangling pointer references
F	Format string vulnerabilities

Table III. Protection level of the countermeasure

Code	Protection level
L	Low assurance
M	Medium assurance
H	High assurance

3.3 Protection level

This property describes the level of protection a countermeasure provides for the vulnerabilities for which it was designed.

3.3.1 *Low assurance countermeasures.* Low assurance countermeasures make exploiting a vulnerability harder; however, a method to bypass this countermeasure has been discovered and is practical. For example, a return-into-libc attack is a practical attack on non-executable memory countermeasures. Another reason for marking a countermeasure as low assurance is because it aims to protect against a specific attack technique, e.g. heap-spraying, but does not prevent other ways of exploiting the vulnerability.

3.3.2 *Medium assurance countermeasures.* Medium assurance countermeasures offer better protection than low assurance countermeasures. As long as the assumptions that the countermeasure was built on are preserved, no way of bypassing these countermeasures is currently known that is practical. However, these assumptions do not always hold in the real world. There may also be a possible way of bypassing these countermeasures which is not practical. An example of countermeasures that fall under this category are canary-based countermeasures that use random numbers for protection. The random number must remain secret, which is an assumption that does not always hold in the real world: attackers may be able to find out the number through memory leaks. An attack may also be able to guess the random number given enough attempts, even though the range of possibilities may be too high to make this immediately practical (e.g. some countermeasures have 2^{32} possible combinations on a 32-bit systems).

3.3.3 *High assurance countermeasures.* High assurance countermeasures offer a high degree of assurance that they will work against a specific vulnerability (e.g. if a countermeasure only targets buffer overflows and has high assurance, it will only have high assurance for buffer overflows). Countermeasures which offer memory safety, type safety or can offer verifiable guarantees will have a high assurance

Table IV. Stage

Code	Stage
Imp	Implementation
Test	Debugging & Testing
Pack	Packaging
Depl	Deployment

Table V. Effort

Code	Effort
Auto	Automatic
ManS	Small manual effort
ManL	Larger manual effort

rating. For example, a safe language that explicitly removes or checks constructs to prevent a specific vulnerability from occurring will have a high assurance protection level. Sometimes weaker countermeasures which still offer a high level of assurance will also fall in this section. For example, a bounds checker which ensures that no object writes outside its bounds will have a high assurance protection level even though it may be possible for an attacker to overwrite a pointer inside a structure via another element in the same structure. This is something many bounds checkers will not detect because this is valid according to the C standard and preventing this type of vulnerability would break valid programs.

3.4 Usability

These properties describe how the countermeasures can be applied. We differentiate between two subproperties: stage and effort.

Stage (Table IV) denotes where in the software engineering process the countermeasure can be applied.

None of the countermeasures in this paper will operate on the requirement, analysis or design stages of a product, so these stages have been left out of the table. Countermeasures that affect the way an application is coded (i.e. safe languages) fall under the implementation stage. Some countermeasures are built for debugging purposes; these fall under the debugging and testing stage. Some countermeasures are compiled into the program, or modify the binary before it is shipped to customers. These countermeasures operate at the packaging stage. Deployment countermeasures are only applied after the program has been shipped to the customer and usually try and protect more than one application (e.g. kernel patches, sandboxes, etc.).

Effort (Table V) describes the amount of effort required to use the countermeasure. We define a countermeasure to be automatic if it requires no further human effort besides applying the countermeasure. Manual countermeasure requires more effort for a countermeasure to be applied (e.g. modification of source code). We also apply a modifier to determine the amount of manual effort required, small or large.

3.5 Limitations

In Table VI we list the category of limitations in applicability of a countermeasure. Some countermeasures are implemented as hardware changes, this can be a limiting factor in being able to apply a countermeasure in general cases. Other countermea-

Table VI. Limitations

Code	Limitations
HW	Hardware (or virtual machine) changes needed
OS	Operating system changes needed
Arch	Architecture or operating system specific
Src	Source code required
Obj	Object code required
Deb	Debugging symbols required
Dyn	Dynamically linked executable required
Stat	Statically linked application required
Inc	Incompatible with existing compiled code
Chg	Possible changes required in source code
False	May suffer from false positives (identifies a program as vulnerable when it's not)

Table VII. Cost

Code	Cost
?	Unknown
0	None
$--$	Very low
$-$	Low
$-+$	Medium
$+$	High
$++$	Very high

sures are implemented as modifications to the operating system. This is sometimes a limiting factor for applying a countermeasure but it can also be a benefit. An OS-based countermeasure should work for all software running on the OS. Some countermeasure require access to the source code or at least to the debugging symbols, so that they can instrument the software that is being protected correctly. In some cases, countermeasures will be incompatible with existing compiled code such as libraries. This is especially the case if they modify binary representations of particular datatypes (like pointers). Safe languages will generally require a programmer to modify his source code. A few of the countermeasures described in this paper rely on specific features of some architectures or operating systems, which makes it unlikely that they could be ported to the other architectures without significant reengineering. A number of countermeasures also suffer from false positives and/or false negatives. However, if the countermeasure is not complete, by definition it will suffer from false negatives. As such, we have only added a property to denote whether false positives are present or not to the limitations.

3.6 Computational and memory cost

Computational and memory cost give an estimate of the run-time cost a specific countermeasure could incur when deployed. The values listed there are provided as-is. In some cases it was extremely hard to determine the cost based on the descriptions given by the authors and as such some values in these columns might not be entirely accurate. The costs range from none to very high for both computational and memory cost.

4. COUNTERMEASURES

In this section we provide a description of the different categories of countermeasures and an overview of the properties of specific countermeasures. At the top level, we distinguish between eight categories based on the main technique that was used to design the countermeasure. Each of these categories is discussed in a separate subsection. We first describe the key ideas behind the category, and the main advantages and disadvantages. Next, we provide a table listing all proposed countermeasures in that category. The table evaluates each of the countermeasures by providing values for each of the properties discussed in the previous section.

4.1 Safe languages

Safe languages are languages where it is generally not possible for one of the previously mentioned vulnerabilities to exist as the language constructs prevent them from occurring. A number of safe languages are available that will prevent the kinds of implementation vulnerabilities discussed in this text entirely. Examples of such languages include Java and ML but these are not in the scope of our discussion since this document focuses on C and C++. Thus we will only discuss safe languages which remain as close to C or C++ as possible: these safe languages are mostly referred to as safe dialects of C. Some dialects will only need minimal programmer intervention to compile programs [Necula et al. 2002], while others require substantial modification [Jim et al. 2002]. Others severely restrict the C language to a subset to make it safer [Kowshik et al. 2002] or will prevent behavior that the C standard marks as undefined [Oiwa et al. 2002].

In an attempt to prevent dangling pointer references, memory management is handled differently by these safe languages. In some cases the programmer is not given explicit control over deallocation anymore. The free operation is either replaced with a no-operation or removed altogether. In the languages described hereafter two types of memory management are used to prevent dangling pointer references:

Garbage collection does not deallocate memory instantaneously, but defers this to a scheduled time interval or till memory constraints require it to be collected. When garbage collection is done, only memory to which no references exist anymore is deallocated, preventing pointers from referring to deallocated memory [Boehm and Weiser 1988]. However, C programs will generally not clear all pointers to a memory location when they free that location. As such, using garbage collection without modifying the program could result in the program using an unacceptable amount of memory. This is an important problem with using garbage collection for C programs.

Region-based memory management deallocates regions as a whole, memory locations can no longer be deallocated separately. A pointer to a memory location can only be dereferenced if the region that this memory address is in is marked "live". Programmers can manually allocate memory in such a region and have it deallocated as a whole. This introduces some problems with objects that live too long, as a result of being placed in a region that remains live very long. As such this type of memory management will usually require garbage collection, to deallocate heap-based objects, which are in one large region. However by ensuring that a pointer is unique upon deallocation (i.e. the pointer has no aliases), a programmer can safely deallocate memory without causing dangling pointer references [Hicks et al. 2004].

Automatic Pool Allocation makes use of a points-to graph to allocate objects that have the same points-to graph node in a same pool in the heap. As a result, all objects of the same type will be allocated in the same pool. This allows programmers to manually allocate and free memory in the heap. While dangling pointer references may occur, they will point to the same type of object in memory. As a result, it is harder to break memory safety when Automatic Pool Allocation is used [Lattner and Adve 2005].

To prevent the other implementation errors that we described in section 2, several techniques are usually combined. Firstly static analysis is performed to determine if a specific construct can be proven safe. If the construct cannot be proven safe, then generally run-time checks are added to prevent these errors at run-time (e.g. if use of a specific array cannot statically be determined to be safe then code that does run-time bounds checking might be added). To aid this static analysis, pointers are often divided into different types (e.g. never-null pointers, which are guaranteed to never contain a NULL value) depending on the amount of checking that should be added at runtime. Some of the safe languages will infer these pointer types automatically, while others expect the programmer to explicitly use new types of pointers.

Table VIII. Safe languages

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Jim et al. [2002]	PD	SHDF	H	Imp	ManL	Src +Chg	− + / − +
Necula et al. [2002]	PD	SHDF	H	Imp	ManS	Src + Chg	+/+
Larus et al. [2004]	PD	SHDF	H	Imp	ManL	Src + Chg	?/?
Dhurjati et al. [2003]	P	SHDF	H	Imp	ManL	Src + Chg	?/?
Oiwa et al. [2002]	PD	SHDF	H	Pack	ManS	Src + Chg	+/+
Xu et al. [2004]	P	SHD	H	Pack	Auto	Src + Chg	+/+
Dhurjati et al. [2006]	PD	SHD	M	Pack	Auto	Src + Chg	− +/+
Condit et al. [2007]	PD	SH	H	Imp	ManL	Src + Chg	− +/+

Table VIII gives an overview of the safe languages we examined. Some global properties can be gathered from the table. None of the languages will support all constructs in C, so some programs need to be modified. The effort required to do these changes varies.

Most languages will work at the implementation level: the programmer either implements his program directly for the language or modifies his program to make it work correctly. However, Fail-safe C [Oiwa et al. 2002] is a C compiler that attempts to make C safer by preventing undefined behavior. As a result, it will not compile all correct programs, but the changes required are limited.

Control-C [Dhurjati et al. 2003; Kowshik et al. 2002] is also of particular interest because it will not add dynamic checks when compiling a program. It restricts the C language to a specific subset and makes a number of assumptions about the runtime system. It is designed to run on the Low Level Virtual Machine (LLVM) system, where all memory and register operations are type safe.

The computational and memory costs of these languages is inversely related to the amount of effort required to port a program to the language: small effort means higher overheads while larger effort means lower overheads.

4.2 Bounds checkers

Full bounds checkers are the best protection against buffer overflows. They will check every array indexation and pointer arithmetic to ensure that they do not attempt to write to or read from a location outside of the space allocated for them. Two important techniques are used to perform traditional full bounds checking: adding bounds information to all pointers or to objects. Originally bounds checkers had significant overhead both in terms of memory and performance, however recent advances have significantly reduced overheads in both these areas without compromising on security [Younan et al. 2010; Dhurjati and Adve 2006a; Akritidis et al. 2009; Nagarakatte et al. 2009].

4.2.1 *Adding bounds information to all pointers.* These bounds checkers store extra information about pointers. Besides the current value of the pointer, they also store the lower and upper bound of the object that the pointer refers to. This can be stored in two ways: by modifying the representation of the pointer itself to include this information (fat pointers) or by storing this bounds information in a table and by looking it up when needed. When the pointer is used, a check will be performed to make sure it will not write beyond the bounds of the object to which it refers. A problem with this approach is that it is incompatible with unprotected code (for example shared libraries). Because the bounds information is stored with the pointer, it must be updated when the pointer changes. However these changes to bounds information can not be tracked through unprotected code, so the value of the pointer may have changed while the bounds checker still has the bounds information for the original object to which the pointer referred. Fat pointers compound the problem because they must be cast to regular pointers when passed to the unprotected code and then later must be cast back to fat pointers. Since the bounds information was stored with the pointer, the bounds checker may have trouble finding the bounds of the object that the pointer is referring, so it may not be able to cast it back to a fat pointer.

4.2.2 *Adding bounds information for objects.* Pointers remain the same, but a table stores the bounds information of all objects. Using the pointer's value, it can be determined what object it is pointing to. Then, pointer arithmetic and/or pointer use is checked: the bounds of the object the pointer refers to is looked up and if the result of pointer arithmetic would make the pointer point outside the bounds of the object, an overflow has been detected.

Table IX. Bounds checkers with pointer bounds information

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Kendall [1983]	PD	SH	H	Pack	Auto	Src	+/+
Steffen [1992]	PD	SH	H	Pack	ManS	Src + Chg	+/+
Austin et al. [1994]	PD	SHD	H	Pack	Auto	Src + Inc	+/+
Patil and Fischer [1997]	PD	SHD	H	Pack	Auto	Src + Chg	++/++
Lam and Chiueh [2005]	D	SH	M	Pack + Depl	Auto	Src + Arch + OS	−/−+
Shao et al. [2006]	PD	SH	H	Pack	Auto	HW + Src + Inc + Arch	−/+
Hiser et al. [2009]	PD	SHD	M	Depl	Auto	False + Stat	++/++

4.2.3 *Other types of bounds checkers.* This category contains other bounds checkers which will do some kind of bounds checking but are different with respect to the traditional checkers in that they do not strive for complete checking of all objects. Some of these last types will ensure that a string manipulation function will only write inside the stack frame that the destination pointer is pointing to [Baratloo et al. 2000; Snarskii 1997] or will ensure that the function does not write past the bounds of the destination string [Avijit et al. 2006].

4.3 Probabilistic countermeasures

Many countermeasures make use of randomness when protecting against the attacks. Many different approaches exist when using randomness for protection. Canary-based countermeasures use a secret random number (the *canary*) that they store before an important memory location. If the random number has changed after some operations have been performed then an attack was detected. Memory-obfuscation countermeasures will encrypt (usually with XOR) important memory locations or other information using random numbers while these are in memory and will decrypt them before they are transferred to registers. Memory layout randomizers will randomize the layout of memory, for instance by loading the stack and heap at random addresses. Some memory layout randomizers will also ensure that objects are spaced out at random intervals from each other, preventing an attacker from knowing exactly how far one object is from another. Instruction set randomizers will encrypt the instructions while in memory and will decrypt them before execution. One important limitation for these approaches is that they will

Table X. Bounds checkers with object bounds information

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Jones and Kelly [1997]	PD	SH	H	Pack	ManS	Src + Chg	+ + / + +
Lhee and Chapin [2002]	PD	SH	M	Pack	Auto	Src	− + /+
Ruwase and Lam [2004]	PD	SH	H	Pack	Auto	Src	+ + / + +
Rinard et al. [2004]	PD	SH	H	Pack	Auto	Src	+ + / + +
Dhurjati and Adve [2006a]	PD	SH	H	Pack	Auto	Src	− + /?
Nethercote and Fitzhardinge [2004]	PD	SHD	L	Depl	Auto	Dyn + Deb	+ + / + +
Rinard et al. [2004]	P	SH	H	Pack	Auto	Src	++/+
Nagarakatte et al. [2010]	PD	SHD	H	Pack	Auto	Src	+ − /?
Nagarakatte et al. [2009]	PD	SHD	H	Pack	Auto	Src	+ − /+
Younan et al. [2010]	PD	SH	H	Pack	Auto	Src	+ − /−
Akritidis et al. [2009]	PD	SH	H	Pack	Auto	Src	+ − /−
Akritidis et al. [2008]	P	SH	M	Pack	Auto	Src	−/−
Hastings and Joyce [1992]	PD	SHD	M	Pack	Auto	-	+ + /+
Chinchani et al. [2004]	PD	SHD	L	Pack	Auto	Src	+/ + −
Arahori et al. [2009]	D	SH	M	Pack	ManS	Src	+ − / − −

Table XI. Other types of bounds checkers

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Avijit et al. [2006]	PD	SH	M	Depl	Auto	Deb + Dyn	− + / − +
Baratloo et al. [2000]	D	S	L	Depl	Auto	Dyn	−/−
Snarskii [1997]	D	S	L	Depl	Auto	Dyn	−/−
Fetzer and Xiao [2001]	D	H	L	Depl	Auto	Dyn	+/+
Li and cker Chiueh [2007]	D	F	M	Depl	Auto	Arch	+ − / − −

mostly rely on the assumption of memory secrecy, i.e., that the application does not leak information, which could allow an attacker to bypass the countermeasure. However, this assumption does not always hold [Strackx et al. 2009].

4.3.1 *Canaries.* The observation that attackers usually try to overwrite the return address when exploiting a buffer overflow led to a string of countermeasures that were designed to protect the return address. One of the earliest examples of this type of protection is the canary-based countermeasure [Cowan et al. 1998]. These countermeasures protect the return address by placing a value before it on the stack that must remain unchanged during program execution. Upon entering a function, the canary is placed on the stack below the return address. When the function is done with executing, the canary stored on the stack will be compared to the original canary. If the stack-stored canary has changed an overflow has occurred and the program can be safely terminated. A canary can be a random number, or a string which is hard to replicate when exploiting a buffer overflow (e.g. a NULL byte). StackGuard [Cowan et al. 1998] was the first countermeasure to use canaries to offer protection against stack-based buffer overflows; however, attackers soon discovered a way of bypassing it using indirect pointer overwriting. Attackers would overwrite a local pointer in a function and make it point to a target location, when the local pointer is dereferenced for writing, the target location is overwritten without modifying the canary (see Section 2.1.2.2 for a more detailed description). Propolice [Etoh and Yoda 2000] is an extension of StackGuard, it fixes these type of attacks by reordering the stack frame so that buffers can no longer overwrite pointers in a function. These two countermeasures have been extremely popular. Propolice has been integrated into the GNU C Compiler and a similar countermeasure has made it's way into Visual Studio's compiler [Bray 2002; Grimes 2004].

Canaries were later also used to protect other memory locations, like the management information of the memory allocator that is often overwritten using a heap-based buffer overflow [Krennmair 2003].

Table XII. Canary-based countermeasures

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Cowan et al. [1998]	D	S	L	Pack	Auto	Src	−/−
Bray [2002]	D	S	M	Pack	Auto	Src	−/−
Etoh and Yoda [2000]	D	S	M	Pack	Auto	Src	−/−
Krennmair [2003]	D	H	M	Depl	Auto	Dyn	−/−
Zuquete [2004]	D	S	M	Pack	Auto	Src	+/−

4.3.2 *Obfuscation of memory addresses.* Memory-obfuscation countermeasures use a closely related approach based on random numbers. These random numbers are used to 'encrypt' specific data in memory and to decrypt it before using it in an execution. These approaches are currently used for obfuscating pointers (XOR with a secret random value) while in memory. When the pointer is later used in an instruction it is first decrypted to a register. If an attacker attempts to overwrite the pointer with a new value, it will have the wrong value when decrypted, which will most likely cause the program to crash. A problem with this approach is that XOR encryption is bytewise encryption. If an attacker only needs to overwrite 1 or 2 bytes instead of the entire pointer, then the chances of guessing the pointer correctly vastly improve (from 1 in 4 billion to 1 in 65000 or even 1 in 256) [Alexander 2005]. If the attacker is able to control a relatively large amount of memory (e.g. with a buffer overflow), then the chances of a successful attack increase even more. While it would be possible to use better encryption, it would likely be prohibitively expensive since every pointer needs to be encrypted and decrypted this way.

Table XIII. Obfuscation-based countermeasures

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Robertson et al. [2003]	D	H	M	Depl	Auto	Dyn	−/−
Frantzen and Shuey [2001]	D	S	L	Depl	Auto	Arch	−/−
Cowan et al. [2003]	M	SHDF	L	Pack	Auto	Src	− + /−
Zhu and Tyagi [2004]	D	SHDF	L	Pack	Auto	Src	−/−
Shao et al. [2003]	D	SH	L	Pack	Auto	HW + Chg	− − /0
Bhatkar and Sekar [2008]	M	SHDF	M	Pack	Auto	Src	+/?
Gadaleta et al. [2010]	D	SHDF	L	Depl	Auto	OS	−/−
Roglia et al. [2009]	M	SHDF	M	Pack	Auto	Dyn	−/−
Liang et al. [2009]	M	SHDF	L	Pack	Auto	Src	− − /0
Tuck et al. [2004]	M	SHDF	M	Pack	Auto	Src + HW	− − / − −
Jiang et al. [2007]	M	SHDF	M	Depl	Auto	Chg	+ + / − −

4.3.3 *Memory Randomization.* Memory randomization is another approach that makes executing injected code harder. Most exploits expect the memory segments to always start at a specific address and attempt to overwrite the return address of a function, or some other interesting address with an address that points into their own code. However for attackers to be able to point to their own code, they must know where in memory their code resides. If the base address is generated randomly when the program is executed, it is harder for the exploit to direct the execution-flow to its injected code because it does not know the address that the injected code is loaded at. This technique is also called address space layout randomization (ASLR). Shacham et al. [2004] examines limitations to the amount of randomness such an approach can use due to address space limitations. The paper also describes a guessing attack that can be used against programs that use forking as these will usually not be rerandomized.

Several approaches will go even further in their approach to randomization, not simply randomizing the base address but also randomizing the amount of space between objects, making it hard for the attacker to figure out the relative distance between objects by adding a layer of difficulty.

Table XIV. Memory randomization-based countermeasures

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
The PaX Team [2000]	M	SHDF	L	Depl	Auto	OS	$--/-$
Xu et al. [2003]	M	SHDF	L	Depl	Auto	-	$-/-+$
Bhatkar et al. [2003]	M	SHDF	L	Depl	Auto	-	$0/-$
Berger and Zorn [2006]	D	H	M	Depl	Auto	Dyn	$-+/-+$
Bhatkar et al. [2005]	M	SHDF	M	Pack + Depl	Auto	Src	$-/0$
Kil et al. [2006]	M	SHDF	M	Depl	Auto	Src + OS	$--/-$
Lin et al. [2009]	M	SHDF	L	Pack	ManS	Src	$-/-$
Kharbutli et al. [2006]	M	H	M	Depl	Auto	Dyn	$-/-+$

4.3.4 *Instruction set randomization.* ISR is another technique that can be used to prevent the injection of attacker-specified code. Instruction set randomization prevents an attacker from injecting any foreign code into the application by encrypting instructions on a per process basis while they are in memory and decrypting them when they are needed for execution. Attackers are unable to guess the decryption key of the current process, so their instructions, after they have been decrypted, will cause the wrong instructions to be executed. This will prevent attackers from having the process execute their payload and will have a large chance crashing the process due to an invalid instruction being executed. However if attackers are able to print out specific locations in memory, they can bypass the countermeasure. Weiss and Barrantes [2006] and Sovarel et al. [2005] discuss more advanced attacks using small loaders to find the encryption key. Two implementations [Barrantes et al. 2003; Kc et al. 2003] examined in this survey incur a significant run-time performance penalty when unscrambling instructions because they are implemented in emulators. It is entirely possible, and in most cases desirable, to implement them at the hardware level thus reducing the impact on run-time performance.

Table XV. Instruction set randomization-based countermeasures

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Hu et al. [2006]	D	SHDF	M	Depl	Auto	HW	$-+/++$
Barrantes et al. [2003]	M	SHDF	L	Depl	Auto	HW	$+/+$
Kc et al. [2003]	M	SHDF	L	Depl	Auto	HW	$-/+$
Kim et al. [2006]	M	SHDF	L	Depl	Auto	HW	$++/+$

None of these countermeasures can offer a complete protection against the code injection attacks described in Section 2, they all rely on the fact that memory must remain secret. If an attacker is able to read out memory through a format string vulnerability or another type of attack, then the countermeasures can be bypassed entirely. However, a major advantage of these approaches is that they have low computational and memory overheads, making them more suited for production environments. A notable exception to the previous limitation is the work by Hu et al. [2006]. In this paper an approach to instruction set randomization is described that makes use of AES encryption for instructions. This can alleviate the risk which is posed by an attacker being able to read memory locations and finding out the key.

4.4 Separators and replicators of information

Countermeasures that rely on separation or replication of information exist in two types: the first type will try to replicate valuable control-flow data or will separate this data from regular data. The second type relies on replication only, but replicates processes with some diversity introduced, if the processes act differently for the same input, then an attack has been detected

4.4.1 *Separators and replicators of data.* Separation or replication of data makes it harder for an attacker to overwrite this information using an overflow. Some countermeasures will simply copy the return address from the stack to a separate stack and will compare it to or replace the return addresses on the regular stack before returning from a function. These countermeasures are easily bypassed using indirect pointer overwriting where an attacker overwrites a different memory location instead of the return address by using a pointer on the stack. More advanced techniques try to separate all control-flow data, like return addresses and pointers, from regular data, making it harder for an attacker to use an overflow to overwrite this type of data.

4.4.2 *Process replicators.* Other countermeasures in this category will replicate processes and introduce some kind of diversity in these applications. They will execute the processes concurrently, providing each of these processes with the same input and expect the programs to behave in the same way. However the replicated processes are diversified in some way (using ASLR, growing the stack upwards instead of downwards in some processes, etc.). If attackers attempt to exploit the processes, the diversity makes it harder for them to exploit all programs with the same input.

4.5 VMM-based countermeasures

VMM-based countermeasures make use of the Virtual Memory Manager which is present in most modern architectures. Memory is grouped in contiguous regions of fixed sizes (4Kb on Intel IA32) called pages. Virtual memory is an abstraction above the physical memory pages that are present in a computer system. It allows a system to address memory pages as if they are contiguous, even if they are stored on physical memory pages which are not. An example of this is the fact that every process in Linux will start at the same address in the virtual address space, even though physcially this is not the case. Another advantage of virtual memory is the fact that all applications seemingly have 4GB of RAM (on 32 bit systems) available, even if the machine does not have that much physical RAM available. This also allows for the concept of swapping, where memory is written to disk when it is not in active use so the physical memory can be reused for active applications. Translation

Table XVI. Countermeasures that separate or replicate data

Name	Type	Vulns	Prot.	Stage	Effort	Lim.	Cost
Vendicator [2000]	M/D	S	L	Pack	Auto	Src	−/−
Chiueh and Hsu [2001]	D	S	L	Pack	Auto	Src	− + / − −
Xu et al. [2002]	M/D	S	L	Depl	Auto	Src / HW	− − / − −
Lee et al. [2003]	D	S	L	Depl	Auto	HW	− − / − −
Baratloo et al. [2000]	D	S	L	Depl	Auto	Dyn	− + /−
Snarskii [1999]	D	S	L	Depl	Auto	Dyn	−/−
Younan et al. [2006a]	M	HD	M	Depl	Auto	Dyn	− − / − +
Younan et al. [2006b]	M	S	M	Pack	Auto	Src	− − / − +
Prasad and cker Chiueh [2003]	D	S	L	Depl	Auto	False	− − / − −
Smirnov and Chiueh [2005]	D	S	L	Pack	Auto	Src	− − /?
Gadaleta et al. [2009]	M/D	S	L	Pack	Auto	HW + Src	− − / − −
Corliss et al. [2004]	D	S	L	Depl	Auto	HW	−/ − −
Shinagawa [2006]	D	S	L	Pack	Auto	Src + Arch	− − / − −
Gupta et al. [2006]	D/M	S	L	Depl	Auto	-	−/ − −
Dahn and Mancoridis [2003]	M	S	L	Pack	Auto	Src	− − / − −
Francillon et al. [2009]	M	S	L	Depl	Auto	HW Arch	− − /−
Kharbutli et al. [2006]	M	H	M	Depl	Auto	Dyn	−/ − +

Table XVII. Counermeasures that replicate processes

Name	Type	Vulns	Prot.	Stage	Effort	Lim.	Cost
Bruschi et al. [2007a]	D	SHDF	M	Pack	Auto	Src	+/+
Cox et al. [2006]	D	SHDF	M	Depl	Auto	False + Inc	+ + /+
Salamat et al. [2009]	D	S	L	Depl	Auto	False + Inc	+/+

of virtual memory to physical memory is handled by a memory management unit (MMU) which is present in most architectures.

Pages can have specific permissions assigned to them: Read, Write and Execute.

Many of the countermeasures in this section will make use of paging permissions or the fact that multiple virtual pages can be mapped onto the same physical page.

Countermeasures in this category can be divided into two subcategories: non-executable memory (NX) and guard-page based countermeasures.

4.5.1 *Non-executable memory-based countermeasures.* These countermeasures will make data memory non-executable. Most operating systems divide process memory into at least a code (also called the text) and data segment. They will mark the code segment as read-only, preventing a program from modifying code that has been loaded from disk into this segment unless the program explicitly requests write permissions for the memory region. Hence, attackers have to inject their code into the data segment of the application. Most applications do not require executable data segments as all their code will be in the code segment. Some countermeasures mark this memory as non-executable, which will make it harder for an attacker to inject code into a running application. A major disadvantage of this approach is that an attacker could use a code injection attack to execute existing code as is the case in a return-into-libc attack. The countermeasures discussed here were a work-around for the fact that Intel mapped the page read permission to the page execute permission, which meant that if a page was readable, it was also executable. This has been remedied on recent versions of the Intel architecture, and non-executable memory is now available as an option in many operating systems.

Table XVIII. Non-executable memory-based countermeasures

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Solar Designer [1997]	M	S	L	Depl	Auto	OS	0/0
The PaX Team [2000]	M	SHDF	L	Depl	Auto	OS	+/0

4.5.2 *Guard-page-based countermeasures.* These countermeasures will use properties of the virtual memory manager to add protection against attacks. Electric Fence [Perens 1987], for example, will allocate each chunk of heap memory on a separate page and will place a guard page (a page without read, write or execute permissions assigned to it) behind it. If the program writes past its bounds it will try to write into the guard page, which will cause the program to be terminated for accessing invalid memory.

Table XIX. Guard-page-based countermeasures

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Perens [1987]	D	H	L	Test	Auto	Dyn	+/+
Dhurjati and Adve [2006b]	D	D	M	Depl	Auto	Src	+ + /− −

4.6 Execution monitors

Execution monitors monitor specific security relevant events (like system calls) and perform specific actions based on what is monitored. Some monitors try to limit the damage a successful attack on a vulnerability could do to the underlying system by limiting the actions a program can perform. Others detect if a program is exhibiting unexpected behavior and will provide alerts if this occurs.

4.6.1 *Policy enforcement.* Policy enforcers are based on the "Principle of Least Privilege" [Saltzer and Schroeder 1975], where an application is only given as much privileges as it needs to be able to complete its task. These countermeasures define a clear policy, some way or another, specifying what an application specifically can and cannot do. Generally, enforcement is done through a reference monitor where an application's access to specific resources (the term resource is used in the broadest sense: a system call, a file, a hardware device, ...) is regulated. An example of such a countermeasure enforces a policy on system calls that the application is allowed to execute, making sure that the application can not execute system calls that it would not normally need. Other attempts to do the same for file accesses change the program's root directory (chroot) and mirror files under this directory structure that the program can access.

4.6.2 *Fault Isolation.* Fault isolation ensures that certain parts of software do not cause a complete system (a program, a collection of programs, the operating system, ...) to fail. The most common way of providing fault isolation is by using address space separation; however, this will cause expensive context switches to occur that incur a significant overhead during execution. Because the modules are in different address spaces, communication between the two modules will also incur a higher overhead. Although some fault isolation countermeasures will not completely protect a program from code injection, the proposed techniques might still be useful if applied with the limitation of what injected code could do in mind (i.e. run-time monitoring as opposed to transforming source or object code).

4.6.3 *Anomaly detection.* Many of the techniques that are used for policy enforcers can be used for anomaly detection. In many cases the execution of system calls is monitored and if they do not correspond to a previously gathered pattern, an anomaly is recorded. Once a threshold for anomalies is reached, the anomaly can be reported and subsequent action can be taken (e.g. the program is terminated or the system call is denied). However, attackers can perform a mimicry attack against anomaly detectors [Wagner and Dean 2001; Parampalli et al. 2008; Kruegel et al. 2005]. These attacks mimic the behavior of the application that is modeled by the anomaly detector. They may be able to get the application in an unsafe state by mimicking the behavior that the detector would expect to be performed before the state is reached, but reaching the state nonetheless. For example, if an application ever performs an *execve*[1] system call in its lifetime, the attacker could easily execute the system calls that the detector would expect to see before executing the *execve* call.

[1]When a program calls the *execve* system call the current process is replaced with a new process (passed as an argument to *execve*) that inherits the permissions of the currently running process.

Table XX. Policy enforcers

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Erlingsson and Schneider [1999]	C	SHDF	L	Depl	ManL	-	+/−
Evans and Twyman [1999]	C	SHDF	L	Depl	ManL	Dyn	+/−
Goldberg et al. [1996]	C	SHDF	M	Depl	ManL	-	− − /0
Provos [2003]	C	SHDF	M	Depl	ManS	OS	+ − /0
Bernaschi et al. [2000]	C	SHDF	M	Depl	ManL	OS	− − /−
Prevelakis and Spinellis [2001]	C	SHDF	M	Depl	Auto	-	− − /0
Kiriansky et al. [2002]	C	SHDF	H	Depl	Auto	-	+/+
Lin et al. [2005]	C	SHDF	L	Depl	Auto	-	−/ − −
Ringenburg and Grossman [2005]	D	F	M	Pack	Auto	Src	− − /0
Yong and Horwitz [2003]	PD	SHD	M	Pack	Auto	Src	+/+
Abadi et al. [2005]	CD	SHDF	H	Pack	Auto	Stat	− + / − −
Castro et al. [2006]	P	SHDF	M	Pack	Auto	Src	+/+
Kc and Keromytis [2005]	C	SHDF	M	Depl	Auto	OS	− − /0
Rajagopalan et al. [2005]	C	SHDF	M	Depl	Auto	OS	− − /0
Patel et al. [2007]	C	SHDF	M	Pack	Auto	HW + Src	− − /?

Table XXI. Fault isolation

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Wahbe et al. [1993]	C	SHDF	L	Depl	Auto	-	−/−
Small [1997]	C	SHDF	M	Depl	Auto	-	+/−
McCamant and Morrisett [2006]	C	SHDF	M	Depl	Auto	-	− + /−

4.7 Hardened libraries

Hardened libraries replace library functions with versions which contain extra checks. An example of these are libraries which offer safer string operations: more checks will be performed to ensure that the copy is in bounds, that the destination string is properly NULL terminated (something strncpy does not do if the string is too large). Other libraries will prevent format strings from containing '%n' in writable memory [Robbins 2001] or will check to ensure that the amount of format specifiers are the same as the amount of arguments passed to the function [Cowan et al.

Table XXII. Anomaly detectors

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Forrest et al. [1996]	D	SHDF	L	Depl	Auto	-	+/+
Sekar et al. [2001]	D	SHDF	L	Depl	Auto	False	+/−+
Wagner and Dean [2001]	D	SHDF	L	Depl	Auto	-	++/−+
Ratanaworabhan et al. [2009]	D	SHDF	L	Depl	Auto	Dyn + False	−+/0
Egele et al. [2009]	D	SHDF	L	Depl	Auto	OS	+/0
Chen et al. [2009]	D	SHDF	L	Depl	Auto	False	++/0
Feng et al. [2003]	D	SHDF	L	Depl	Auto	False	+/+
Nanda et al. [2006]	D	SHDF	L	Depl	Auto	-	+−/?
Bruschi et al. [2007b]	M	SHDF	M	Depl	Auto	OS + Arch	−−/−−
Ikebe et al. [2008]	D	SHDF	L	Depl	Auto	Deb	+−/+−
Rabek et al. [2003]	D	SHDF	L	Depl	Auto	False	−−/−−

2001].

Table XXIII. Hardened Libraries

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Miller and de Raadt [1999]	P	SH	L	Imp	ManL	Src	0/0
Messier and Viega [2003] (SafeStr)	P	SHF	L	Imp	ManL	Src	?/−
Cowan et al. [2001] (FormatGuard)	D	F	M	Depl	Auto	-	−−/0
Robbins [2001] (Libformat)	D	F	L	Depl	Auto	-	−−/0
Kohli and Bruhadeshwar [2008]	D	F	M	Depl	Auto	Dyn	−−/−−

4.8 Runtime taint trackers

Taint tracking will track information throughout the program: input is considered untrusted and thus tainted. This taint information is tracked throughout the program, if a value is based on tainted information, it becomes tainted. When an application uses tainted information in a location where it expects untainted data, an error is reported. Taint tracking can be used to detect the vulnerabilities described in Section 2. These taint trackers will instrument the program to mark input as tainted. If such tainted data is later used to modify a trusted memory location (like a return address), then a fault is generated.

Table XXIV. Runtime taint trackers

Name	Type	Vulns	Prot.	Stage	Effort	Lim	Cost
Newsome and Song [2005]	D	SHD	M	Depl	Auto	False	++/+
Chen et al. [2005]	D	SHDF	M	Depl	Auto	HW + OS + False	?/?
Xu et al. [2006]	D	SHDF	M	Pack	Auto	Src + False	+/?
Suh et al. [2004]	D	SHF	M	Depl	Auto	HW + False	−−/−−
Lin et al. [2010]	D	SHDF	L	Pack	Auto	Src + OS	+−/+−
Qin et al. [2006]	D	SHDF	M	Depl	Auto	False	+/?
Cavallaro and Sekar [2008]	D	SHDF	M	Pack	Auto	Src + Chg + False	+/?
Dalton et al. [2008]	D	SHDF	M	Depl	Auto	HW + False	−−/+
Ho et al. [2006]	D	SHDF	M	Depl	Auto	OS + False	+/?

One important limitation with these taint trackers is that they suffer from false positives. They could potentially mark correct, safe code as vulnerable. However, most of the countermeasures described here had few to no false positives in the tests performed by the designers of the countermeasures. It is however an important limitation in cases where no false positives can be tolerated.

5. CONCLUSION

Many countermeasures have been designed since the first stack-based buffer overflows appeared. The first countermeasures were designed to specifically stop simple stack-based buffer overflow attacks that would overwrite the return address and execute injected code. Attackers soon discovered new ways of bypassing these simple countermeasures. New types of attacks were also discovered. More complex countermeasures were then designed to protect against these new attacks. This led to a classic arms race between attackers and countermeasure designers.

In this paper we presented an overview of the most commonly exploited vulnerabilities that lead to code injection attacks. More importantly however, we also presented a survey of the many countermeasures that exist to protect against these vulnerabilities together with a framework for evaluating and classifying them. We described the many techniques used to build countermeasures and discussed some of their advantages and disadvantages. We also assigned specific properties to each of the countermeasures that allow the reader to evaluate which countermeasures could be most useful in a given context.

Although we tried to be as complete as possible when discussing the different countermeasures that exist, it can never be entirely complete. Countermeasure design is an active field, so this paper can only provide a snapshot of the current state of the field with respect to specific countermeasures. However, we believe we have provided a strong framework that can be applied to future countermeasures to further evaluate and classify these new countermeasures.

REFERENCES

Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. 2005. Control-flow integrity. In *12th ACM Conference on Computer and Communications Security.*

Akritidis, P., Cadar, C., Raiciu, C., Costa, M., and Castro, M. 2008. Preventing memory error exploits with wit. In *IEEE Symposium on Security and Privacy.*

Akritidis, P., Costa, M., Castro, M., and Hand, S. 2009. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *18th USENIX Security Symposium.*

Aleph One. 1996. Smashing The Stack For Fun And Profit. *Phrack 49.*

Alexander, S. 2005. Defeating compiler-level buffer overflow protection. *;login: The USENIX Magazine 30,* 3.

anonymous. 2001. Once upon a free(). *Phrack 57.*

Arahori, Y., Gondow, K., and Maejima, H. 2009. Tcbc: Trap caching bounds checking for c. In *8th IEEE International Conference on Dependable, Autonomic and Secure Computing.*

Austin, T. M., Breach, S. E., and Sohi, G. S. 1994. Efficient detection of all pointer and array access errors. In *ACM Conference on Programming Language Design and Implementation.*

Avijit, K., Gupta, P., and Gupta, D. 2006. Binary rewriting and call interception for efficient runtime protection against buffer overflows: Research articles. *Software – Practice & Experience 36,* 9.

Baratloo, A., Singh, N., and Tsai, T. 2000. Transparent Run-Time Defense Against Stack Smashing Attacks. In *USENIX Annual Technical Conference Proceedings.*

Barrantes, E. G., Ackley, D. H., Forrest, S., Palmer, T. S., Stefanović, D., and Zovi, D. D. 2003. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *10th ACM Conference on Computer and Communications Security.*

BBP. 2003. BSD heap smashing.

Berger, E. D. and Zorn, B. G. 2006. Diehard: probabilistic memory safety for unsafe languages. In *ACM conference on Programming language design and implementation.*

Bernaschi, M., Gabrielli, E., and Mancini, L. V. 2000. Operating system enhancements to prevent the misuse of system calls. In *7th ACM conference on Computer and communications security.*

Bhatkar, S., DuVarney, D. C., and Sekar, R. 2003. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *12th USENIX Security Symposium.*

Bhatkar, S. and Sekar, R. 2008. Data space randomization. In *5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment.*

Bhatkar, S., Sekar, R., and DuVarney, D. C. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium.*

Boehm, H. and Weiser, M. 1988. Garbage collection in an uncooperative environment. *Software, Practice and Experience 18,* 9.

Bray, B. 2002. Compiler Security Checks In Depth.

Bruschi, D., Cavallaro, L., and Lanzi, A. 2007a. Diversified process replicae for defeating memory error exploits. In *3rd International Workshop on Information Assurance.*

Bruschi, D., Cavallaro, L., and Lanzi, A. 2007b. An efficient technique for preventing mimicry and impossible paths execution attacks. In *3rd International Workshop on Information Assurance.*

Bulba and Kil3r. 2000. Bypassing Stackguard and Stackshield. *Phrack 56.*

Castro, M., Costa, M., and Harris, T. 2006. Securing software by enforcing data-flow integrity. In *7th symposium on Operating Systems Design and Implementation.*

Cavallaro, L. and Sekar, R. 2008. Anomalous taint detection. In *11th International Symposium on Recent Advances in Intrusion Detection*.

Chen, K. and Wagner, D. 2007. Large-scale analysis of format string vulnerabilities in debian linux. In *Workshop on Programming languages and analysis for security*.

Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. 2009. Drop: Detecting return-oriented programming malicious code. In *5th International Conference on Information Systems Security*.

Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., and Iyer, R. K. 2005. Defeating memory corruption attacks via pointer taintedness detection. In *International Conference on Dependable Systems and Networks*.

Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. 2005. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium*.

Chinchani, R., Iyer, A., Jayaraman, B., and Upadhyaya, S. 2004. Archerr: Runtime environment driven program safety. In *9th European Symposium on Research in Computer Security*.

Chiueh, T. and Hsu, F. 2001. RAD: A compile-time solution to buffer overflow attacks. In *21st International Conference on Distributed Computing Systems*.

Condit, J., Harren, M., Anderson, Z., Gay, D., and Necula, G. 2007. Dependent types for low-level programming. In *16th European Symposium on Programming*.

Conover, M. 1999. w00w00 on Heap Overflows.

Corliss, M., Lewis, E. C., and Roth, A. 2004. Using dise to protect return addresses from attack. In *Workshop on Architectural Support for Security and Anti-Virus*.

Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., and Lokier, J. 2001. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *10th USENIX Security Symposium*.

Cowan, C., Beattie, S., Johansen, J., and Wagle, P. 2003. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *12th USENIX Security Symposium*.

Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Symposium*.

Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., and Hiser, J. 2006. N-variant systems: a secretless framework for security through diversity. In *15th USENIX Security Symposium*.

Dahn, C. and Mancoridis, S. 2003. Using program transformation to secure c programs against buffer overflows. In *10th Working Conference on Reverse Engineering*.

Dalton, M., Kannan, H., and Kozyrakis, C. 2008. Real-world buffer overflow protection for userspace & kernelspace. In *17th Usenix Security symposium*.

Dhurjati, D. and Adve, V. 2006a. Backwards-compatible array bounds checking for c with very low overhead. In *28th international conference on Software engineering*.

Dhurjati, D. and Adve, V. 2006b. Efficiently detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks*.

Dhurjati, D., Kowshik, S., and Adve, V. 2006. Safecode: enforcing alias analysis for weakly typed languages. In *ACM conference on Programming language design and implementation*.

Dhurjati, D., Kowshik, S., Adve, V., and Lattner, C. 2003. Memory Safety Without Runtime Checks or Garbage Collection. In *ACM Conference on Language, Compiler, and Tool Support for Embedded Systems*.

Dobrovitski, I. 2003. Exploit for CVS double free() for Linux pserver.

Egele, M., Wurzinger, P., Kruegel, C., and Kirda, E. 2009. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

Erlingsson, U. and Schneider, F. B. 1999. SASI Enforcement of Security Policies: A Retrospective. In *New Security Paradigm Workshop*.

Erlingsson, U., Younan, Y., and Piessens, F. 2010. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer.

Etoh, H. and Yoda, K. 2000. Protecting from stack-smashing attacks. Tech. rep., IBM Research Divison, Tokyo Research Laboratory.

Evans, D. and Twyman, A. 1999. Flexible Policy-Directed Code Safety. In *IEEE Symposium on Security and Privacy*.

Feng, H. H., Kolesnikov, O. M., Fogla, P., Lee, W., and Gong, W. 2003. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*.

Fetzer, C. and Xiao, Z. 2001. Detecting Heap Smashing Attacks Through Fault Containment Wrappers. In *20th IEEE Symposium on Reliable Distributed Systems*.

Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. 1996. A Sense of Self for Unix Processes. In *IEEE Symposium on Security and Privacy*.

Francillon, A. and Castelluccia, C. 2008. Code injection attacks on harvard-architecture devices. In *15th ACM conference on Computer and communications security*.

Francillon, A., Perito, D., and Castelluccia, C. 2009. Defending embedded systems against control flow attacks. In *1st ACM workshop on Secure execution of untrusted code*.

Frantzen, M. and Shuey, M. 2001. StackGhost: Hardware Facilitated Stack Protection. In *10th USENIX Security Symposium*.

Gadaleta, F., Younan, Y., Jacobs, B., Joosen, W., De Neve, E., and Beosier, N. 2009. Instruction-level countermeasures against stack-based buffer overflow attacks. In *1st Workshop on Virtualization Technology for Dependable Systems*.

Gadaleta, F., Younan, Y., and Joosen, W. 2010. Bubble: A javascript engine level countermeasure against heap-spraying attacks. In *2nd International Symposium on Engineering Secure Software and Systems*.

Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. 1996. A Secure Environment for Untrusted Helper Applications. In *6th USENIX Security Symposium*.

Grimes, R. 2004. Preventing Buffer Overflows in C++. *Dr Dobb's Journal: Software Tools for the Professional Programmer 29,* 1.

Gupta, S., Pratap, P., Saran, H., and Arun-Kumar, S. 2006. Dynamic code instrumentation to detect and recover from return address corruption. In *International workshop on Dynamic systems analysis*.

Hastings, R. and Joyce, B. 1992. Purify: Fast Detection of Memory Leaks and Access Errors. In *Winter USENIX conference*.

Hicks, M., Morrisett, G., Grossman, D., and Jim, T. 2004. Experience with safe manual memory-management in cyclone. In *4th international symposium on Memory management*.

Hiser, J. D., Coleman, C. L., Co, M., and Davidson, J. W. 2009. Meds: The memory error detection system. In *1st International Symposium on Engineering Secure Software and Systems*.

Ho, A., Fetterman, M., Clark, C., Warfield, A., and Hand, S. 2006. Practical taint-based protection using demand emulation. In *1st ACM European conference on Computer systems*.

Howard, M. and LeBlanc, D. 2001. *Writing Secure Code*. Microsoft Press.

Hu, W., Hiser, J., Williams, D., Filipi, A., Davidson, J. W., Evans, D., Knight, J. C., Nguyen-Tuong, A., and Rowanhill, J. 2006. Secure and practical defense against code-injection attacks using software dynamic translation. In *2nd international conference on Virtual execution environments*.

Ikebe, Y., Nakayama, T., Katagiri, M., Kawasaki, S., Abe, H., Shinagawa, T., and Kato, K. 2008. Efficient anomaly detection system for mobile handsets. In *2nd International Conference on Emerging Security Information, Systems and Technologies*.

Intel Corporation 2001. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel Corporation. Order Nr 245470.

ISO. 1999. Iso/iec 9899:1999: Programming languages – c. Tech. rep., International Organization for Standards.

Jiang, X., Wang, H. J., Xu, D., and Wang, Y. M. 2007. Randsys: Thwarting code injection attacks with system service interface randomization. In *26th IEEE International Symposium on Reliable Distributed Systems*.

JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. 2002. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*.

JONES, R. W. M. AND KELLY, P. H. J. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *3rd International Workshop on Automatic Debugging*.

KAEMPF, M. 2001. Vudo - an object superstitiously believed to embody magical powers. *Phrack 57*.

KC, G. S. AND KEROMYTIS, A. D. 2005. e-nexsh: Achieving an effectively non-executable stack and heap via system-call policing. In *21st Annual Computer Security Applications Conference*.

KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. 2003. Countering Code-Injection Attacks With Instruction-Set Randomization. In *10th ACM Conference on Computer and Communications Security*.

KENDALL, S. C. 1983. Bcc: Runtime Checking for C Programs. In *USENIX Summer Conference*.

KHARBUTLI, M., JIANG, X., SOLIHIN, Y., VENKATARAMANI, G., AND PRVULOVIC, M. 2006. Comprehensively and efficiently protecting the heap. In *12th international conference on Architectural support for programming languages and operating systems*.

KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. 2006. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *22nd Annual Computer Security Applications Conference*.

KIM, D. J., KIM, T. H., KIM, J., AND HONG, S. J. 2006. Return address randomization scheme for annuling data-injection buffer overflow attacks. In *2nd Confernce on Information Security and Cryptology*.

KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. 2002. Secure Execution Via Program Shepherding. In *11th USENIX Security Symposium*.

KOHLI, P. AND BRUHADESHWAR, B. 2008. Formatshield: A binary rewriting defense against format string attacks. In *13th Australasian conference on Information Security and Privacy*.

KOWSHIK, S., DHURJATI, D., AND ADVE, V. 2002. Ensuring code safety without runtime checks for real-time control systems. In *International Conference on Compilers Architecture and Synthesis for Embedded Systems*.

KRENNMAIR, A. 2003. ContraPolice: a libc Extension for Protecting Applications from Heap-Smashing Attacks.

KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. 2005. Automating mimicry attacks using static binary analysis. In *14th USENIX Security Symposium*.

LAM, L. AND CHIUEH, T. 2005. Checking array bound violation using segmentation hardware. In *International Conference on Dependable Systems and Networks*.

LARUS, J. R., BALL, T., DAS, M., DELINE, R., FÄHNDRICH, M., PINCUS, J., RAJAMANI, S. K., AND VENKATAPATHY, R. 2004. Righting Software. *IEEE Software 21,* 3 (May).

LATTNER, C. AND ADVE, V. 2005. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *ACM Conference on Programming Language Design and Implementation*.

LEE, R. B., KARIG, D. K., MCGREGOR, J. P., AND SHI, Z. 2003. Enlisting Hardware Architecture to Thwart Malicious Code Injection. In *1st International Conference on Security in Pervasive Computing*.

LHEE, K.-S. AND CHAPIN, S. J. 2002. Type-Assisted Dynamic Buffer Overflow Detection. In *11th USENIX Security Symposium*.

LI, W. AND CKER CHIUEH, T. 2007. Automated format string attack prevention for win32/x86 binaries. In *23rd Annual Computer Security Applications Conference*.

LIANG, Z., LIANG, B., AND LI, L. 2009. A system call randomization based method for countering code-injection attacks. *International Journal of Information Technology and Computer Science 1,* 1.

LIN, C., RAJAGOPALAN, M., BAKER, S., COLLBERG, C., DEBRAY, S., AND HARTMAN, J. 2005. Protecting against unexpected system calls. In *14th USENIX Security Symposium*.

Lin, Y.-D., Wu, F.-C., Lai, Y.-C., Huang, T.-Y., and Lin, F. 2010. Embedded tainttracker: Lightweight tracking of taint data against buffer overflow attacks. In *International Conference on Communications.*

Lin, Z., Riley, R. D., and Xu, D. 2009. Polymorphing software by randomizing data structure layout. In *6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.*

McCamant, S. and Morrisett, G. 2006. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium.*

Messier, M. and Viega, J. 2003. Safe c string library v1.0.2.

Microsoft. 2003. Buffer Overrun In RPC Interface Could Allow Code Execution.

Miller, T. C. and de Raadt, T. 1999. strlcpy and strlcat – consistent, safe string copy and concatenation. In *USENIX Annual Technical Conference.*

Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. 2009. Softbound: highly compatible and complete spatial memory safety for c. In *ACM conference on Programming language design and implementation.*

Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. 2010. Cets: compiler enforced temporal safety for c. In *International symposium on Memory management.*

Nanda, S., Li, W., Lam, L.-C., and Chiueh, T.-c. 2006. Foreign code detection on the windows/x86 platform. In *22nd Annual Computer Security Applications Conference.*

Necula, G., McPeak, S., and Weimer, W. 2002. CCured: Type-Safe Retrofitting of Legacy Code. In *29th Symposium on Principles of Programming Languages.*

Nethercote, N. and Fitzhardinge, J. 2004. Bounds-Checking Entire Programs without Recompiling. In *2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management.*

Newsome, J. and Song, D. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *12th Annual Network and Distributed System Security Symposium.*

Oiwa, Y., Sekiguchi, T., Sumii, E., and Yonezawa, A. 2002. Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure: Progress Report. In *International Symposium on Software Security 2002.*

Parampalli, C., Sekar, R., and Johnson, R. 2008. A practical mimicry attack against powerful system-call monitors. In *ACM symposium on Information, computer and communications security.*

Patel, K., Parameswaran, S., and Shee, S. L. 2007. Ensuring secure program execution in multiprocessor embedded systems: a case study. In *5th IEEE/ACM international conference on Hardware/software codesign and system synthesis.*

Patil, H. and Fischer, C. N. 1997. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software: Practice and Experience 27,* 1.

Perens, B. 1987. Electric fence.

Pozza, D., Sisto, R., Durante, L., and Valenzano, A. 2006. Comparing lexical analysis tools for buffer overflow detection in network software. In *1st International Conference on Communication System software and Middleware.*

Prasad, M. and cker Chiueh, T. 2003. A binary rewriting defense against stack based overflow attacks. In *USENIX Annual Technical Conference.*

Prevelakis, V. and Spinellis, D. 2001. Sandboxing Applications. In *USENIX Annual Technical Conference.*

Provos, N. 2003. Improving Host Security with System Call Policies. In *12th USENIX Security Symposium.*

Qin, F., Wang, C., Li, Z., Kim, H.-s., Zhou, Y., and Wu, Y. 2006. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *39th Annual IEEE/ACM International Symposium on Microarchitecture.*

RABEK, J. C., KHAZAN, R. I., LEWANDOWSKI, S. M., AND CUNNINGHAM, R. K. 2003. Detection of injected, dynamically generated, and obfuscated malicious code. In *ACM workshop on Rapid Malcode*.

RAJAGOPALAN, M., HILTUNEN, M., JIM, T., AND SCHLICHTING, R. 2005. Authenticated system calls. In *International Conference on Dependable Systems and Networks*.

RATANAWORABHAN, P., LIVSHITS, B., AND ZORN, B. 2009. Nozzle: A defense against heap-spraying code injection attacks. In *18th Usenix Security Symposium*.

RINARD, M., CADAR, C., DUMITRAN, D., ROY, D., LEU, T., AND BEEBEE, W. S. 2004. Enhancing server availability and security through failure-oblivious computing. In *6th Symposium on Operating Systems Design and Implementation*.

RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., AND LEU, T. 2004. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *20th Annual Computer Security Applications Conference*.

RINGENBURG, M. F. AND GROSSMAN, D. 2005. Preventing format-string attacks via automatic and efficient dynamic checking. In *12th ACM conference on Computer and communications security*.

RIX. 2000. Smashing C++ VPTRs. *Phrack 56*.

ROBBINS, T. 2001. Libformat.

ROBERTSON, W., KRUEGEL, C., MUTZ, D., AND VALEUR, F. 2003. Run-time Detection of Heap-based Overflows. In *17th Large Installation Systems Administrators Conference*.

ROGLIA, G. F., MARTIGNONI, L., PALEARI, R., AND BRUSCHI, D. 2009. Surgically returning to randomized lib(c). In *25th Annual Computer Security Applications Conference*.

RUWASE, O. AND LAM, M. S. 2004. A Practical Dynamic Buffer Overflow Detector. In *11th Annual Network and Distributed System Security Symposium*.

SALAMAT, B., JACKSON, T., GAL, A., AND FRANZ, M. 2009. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *4th ACM European conference on Computer systems*.

SALTZER, J. H. AND SCHROEDER, M. D. 1975. The Protection of Information in Computer Systems. *Proceedings of the IEEE 63,* 9.

SCUT. 2001. Exploiting format string vulnerabilities.

SEKAR, R., BENDRE, M., DHURJATI, D., AND BOLLINENI, P. 2001. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*.

SHACHAM, H. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM conference on Computer and Communications Security*.

SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. 2004. On the Effectiveness of Address-Space Randomization. In *11th ACM conference on Computer and communications security*.

SHAO, Z., CAO, J., CHAN, K. C. C., XUE, C., AND SHA, E. H.-M. 2006. Hardware/software optimization for array & pointer boundary checking against buffer overflow attacks. *Journal of Parallel and Distributed Computing 66,* 9.

SHAO, Z., ZHUGE, Q., HE, Y., AND SHA, E. H. M. 2003. Defending embedded systems against buffer overflow via hardware/software. In *19th Annual Computer Security Applications Conference*.

SHINAGAWA, T. 2006. Segmentshield: Exploiting segmentation hardware for protecting against buffer overflow attacks. In *25th IEEE Symposium on Reliable Distributed Systems*.

SKYLINED. 2004. Internet explorer iframe src&name parameter bof remote compromise.

SMALL, C. 1997. A Tool For Constructing Safe Extensible C++ Systems. In *3rd USENIX Conference on Object-Oriented Technologies*.

SMIRNOV, A. AND CHIUEH, T. 2005. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *Network and Distributed System Security Symposium*.

SMITH, N. P. 1997. Stack Smashing Vulnerabilities In The Unix Operating System.

SNARSKII, A. 1997. Freebsd libc stack integrity patch.

SNARSKII, A. 1999. Libparanoia.

Solar Designer. 1997. Non-executable stack patch.

Solar Designer. 2000. JPEG COM Marker Processing Vulnerability in Netscape Browsers.

Sovarel, A. N., Evans, D., and Paul, N. 2005. Where's the feeb? the effectiveness of instruction set randomization. In *14th conference on USENIX Security Symposium*.

Spafford, E. H. 1989. Crisis and Aftermath. *Communications of the ACM 32,* 6.

Steffen, J. L. 1992. Adding Run-Time Checking to the Portable C Compiler. *Software: Practice and Experience 22,* 4.

Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T. 2009. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security*.

Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S. 2004. Secure program execution via dynamic information flow tracking. In *11th international conference on Architectural support for programming languages and operating systems*.

The PaX Team. 2000. Documentation for the PaX project.

Tuck, N., Calder, B., and Varghese, G. 2004. Hardware and binary modification support for code pointer protection from buffer overflow. In *37th annual IEEE/ACM International Symposium on Microarchitecture*.

Vendicator. 2000. Documentation for Stack Shield.

Viega, J. and McGraw, G. 2002. *Building Secure Software*. Addison-Wesley.

Wagner, D. and Dean, D. 2001. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy*.

Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. 1993. Efficient Software-Based Fault Isolation. In *14th ACM Symposium on Operating System Principles*.

Weiss, Y. and Barrantes, E. G. 2006. Known/chosen key attacks against software instruction set randomization. In *22nd Annual Computer Security Applications Conference*.

Wilander, J. and Kamkar, M. 2002. A Comparison of Publicly Available Tools for Static Intrusion Prevention. In *7th Nordic Workshop on Secure IT Systems*. Karlstad.

Wojtczuk, R. 1998. Defeating Solar Designer's Non-executable Stack Patch.

Xu, J., Kalbarczyk, Z., and Iyer, R. K. 2003. Transparent Runtime Randomization for Security. In *22nd International Symposium on Reliable Distributed Systems*.

Xu, J., Kalbarczyk, Z., Patel, S., and Ravishankar, K. I. 2002. Architecture support for defending against buffer overflow attacks. In *2nd Workshop on Evaluating and Architecting System Dependability*.

Xu, W., Bhatkar, S., and Sekar, R. 2006. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*.

Xu, W., DuVarney, D. C., and Sekar, R. 2004. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *12th ACM International Symposium on Foundations of Software Engineering*.

Yong, S. H. and Horwitz, S. 2003. Protecting C programs from attacks via invalid pointer dereferences. In *9th European software engineering conference*.

Younan, Y. 2003. An overview of common programming security vulnerabilities and possible solutions. M.S. thesis, Vrije Universiteit Brussel.

Younan, Y. 2008. Efficient countermeasures for software vulnerabilities due to memory management errors. Ph.D. thesis, Katholieke Universiteit Leuven.

Younan, Y., Joosen, W., and Piessens, F. 2006a. Efficient protection against heap-based buffer overflows without resorting to magic. In *8th International Conference on Information and Communication Security*.

Younan, Y., Joosen, W., and Piessens, F. 2006b. Extended protection against stack smashing attacks without performance loss. In *22nd Annual Computer Security Applications Conference*.

Younan, Y., Joosen, W., Piessens, F., and den Eynden, H. V. 2010. Improving memory management security for c and c++. *International Journal of Secure Software Engineering 2,* 1.

Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., and Joosen, W. 2010. PAriCheck: an efficient pointer arithmetic checker for c programs. In *ACM Symposium on Information, Computer and Communications Security*.

Younan, Y., Philippaerts, P., Piessens, F., Joosen, W., Lachmund, S., and Walter, T. 2009. Filter-resistant code injection on arm. In *16th ACM conference on Computer and communications security*.

Zhu, G. and Tyagi, A. 2004. Protection against indirect overflow attacks on pointers. In *2nd IEEE International Information Assurance Workshop*.

Zuquete, A. 2004. Stackfences: A run-time approach for detecting stack overflows. In *1st International Conference on E-Business and Telecommunication Networks*.

Received 01 2005; revised 05 2007; revised 07 2010; accepted Month Years