# Code Injection in C and C++ : A Survey of Vulnerabilities and Countermeasures

*Yves Younan*
*Wouter Joosen*
*Frank Piessens*

*Report CW 386, July 2004*


Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Code Injection in C and C++ : A Survey of Vulnerabilities and Countermeasures

*Yves Younan*
*Wouter Joosen*
*Frank Piessens*

*Report CW 386, July 2004*

Department of Computer Science, K.U.Leuven

**Abstract**

Implementation errors relating to memory-safety are the most common vulnerabilities used by attackers to gain control over the execution-flow of an application. By carefully crafting an exploit for these vulnerabilities, attackers can make an application transfer execution-flow to code that they have injected. Such code injection attacks are among the most powerful and common attacks against software applications.

This report documents possible vulnerabilities in C and C++ applications that could lead to situations that allow for code injection and describes the techniques generally used by attackers to exploit them.

A fairly large number of defense techniques have been described in literature. An important goal of this report is to give a comprehensive survey of all available preventive and defensive countermeasures that either attempt to eliminate specific vulnerabilities entirely or attempt to combat their exploitation.

Finally, the report presents a synthesis of this survey that allows the reader to weigh the advantages and disadvantages of using a specific countermeasure as opposed to using another more easily.

**CR Subject Classification :** K6.5, A.1, D3

# Contents

# 1 Introduction

Software vulnerabilities are currently, and have been since the advent of multiuser and networked computing, a major cause of computer security incidents [83, 118, 141]. Most of these software vulnerabilities can be traced back to a few mistakes that programmers make over and over again [92]. Even though many documents and books [59, 93, 125, 131] exist that 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 [115]. This document focuses on a specific subclass of software vulnerabilities: implementation errors in C [66] and C++ [37, 120] 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 report.

Several preventive and defensive countermeasures have been proposed to combat exploitation of common implementation errors and this document examines many of these. Our main goal has been to provide a complete survey of all existing countermeasures. We also describe several ways in which some of the proposed countermeasures can be circumvented. Although some countermeasures examined here protect against the more general case of buffer overflows, this document focuses on examining protection against attacks where an attacker specifically attempts to execute code that an application would not execute in normal circumstances (e.g. injecting code, calling a library with specific arguments).

What follows is a general overview of the types of countermeasures that are examined in this document:

**Safe languages** are languages in which most of the discussed implementation vulnerabilities have been made hard or impossible. 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.

**Static source code analyzers** attempt to find implementation vulnerabilities by analyzing the source code of an application, this could be as simple as looking for library functions known to be vulnerable to an implementation error (e.g. ITS4 [124]) or as complicated as making a full model of the program (e.g. PREfix [19, 74]) and then deciding what constructs might cause a specific vulnerability.

**Dynamic analysis and testing tools** instrument the program to generate specific events when an error is encountered.

**Sandboxing** does not attempt to prevent a vulnerability or even to prevent its abuse, it attempts to contain the damage that the abuse of a vulnerability might precipitate.

**Anomaly detection** tries to detect attacks by trying to find out whether an application is showing irregular behavior by attempting to execute system calls it is not supposed to execute at a particular moment during the execution of the program.

**Compiler modifications** modify the way in which a compiler generates code, mostly by adding sanity checks or code that protects critical memory addresses.

**Operating system and hardware modifications** attempt to make injection of foreign code into a running application harder or impossible.

**Library modifications** cover a wide range of possible modifications, they are generally used to easily make changes to existing dynamically linked programs without the need for recompilation: library countermeasures range from just replacing functions that are often misused to linking a library that will change the way a program behaves at run-time.

This document is structured as follows: section 2 contains an overview of the implementation errors that the countermeasures in section 3 attempt to defend against. It also describes typical ways in which these implementation errors can be abused. Section 3 contains our survey of the countermeasures for these vulnerabilities and in some cases, ways in which they can be circumvented and suggestions on how to prevent this. Section 4 presents a synthesis of these results which allows the reader to weigh the advantages and disadvantages of using a specific countermeasures as opposed to using another more easily. Section 5 examines related work in the field of vulnerability and countermeasure surveys. Section 6 discusses our plans for future work and 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 mention 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 [140].

When we describe the exploitation techniques in this section we focus mostly on the IA32-architecture [60]. While the details for exploiting a specific vulnerabilities are architecture dependent, the main techniques presented here should be applicable to other architectures as well.

Finally, we give a short overview of the techniques that are used to defend against the specific vulnerability being described. A more detailed discussion of these techniques is then presented in the survey in section 3.

## 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 run-time 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 at 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. 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

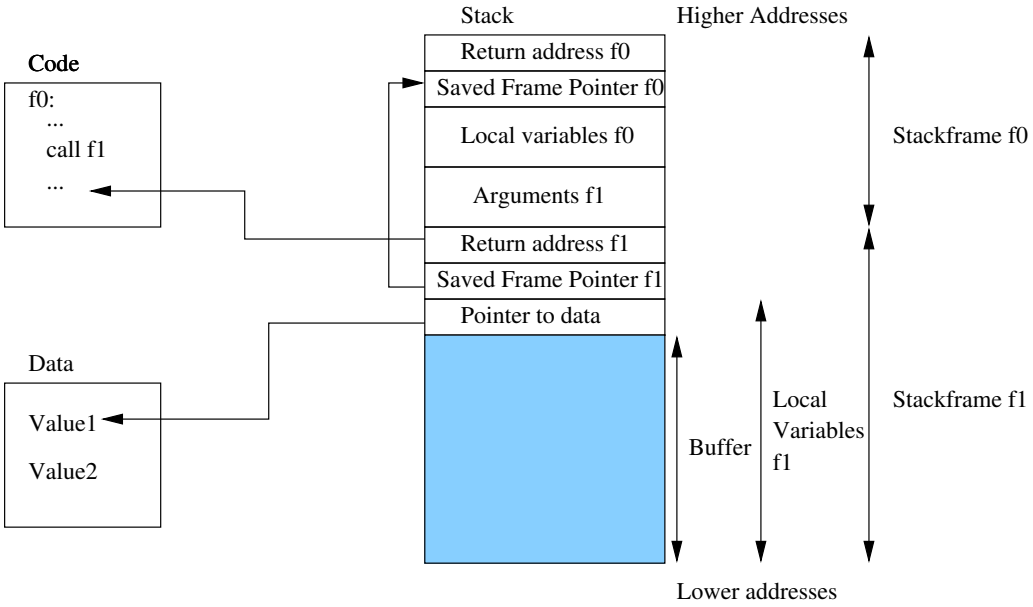


Figure 1: Stack-layout on the IA32

normal. The colored part indicates what could be written to by the function if the buffer is used correctly.

### 2.1.2 Exploitation

#### 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 transfered there [87, 112].

#### Frame pointer overwriting

In some cases it might not be possible for attackers to overwrite the return address. This could be because they can only overwrite a few bytes and can't reach the return address or because the return address has been protected by some countermeasure. Figure 3 shows how an attacker could manipulate the

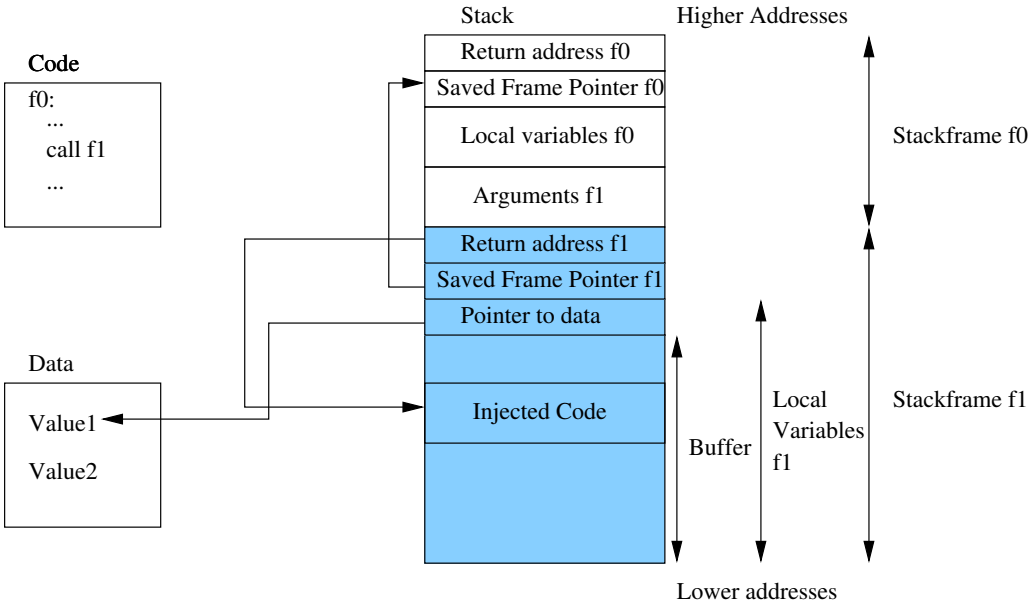


Figure 2: Normal stack-based buffer overflow

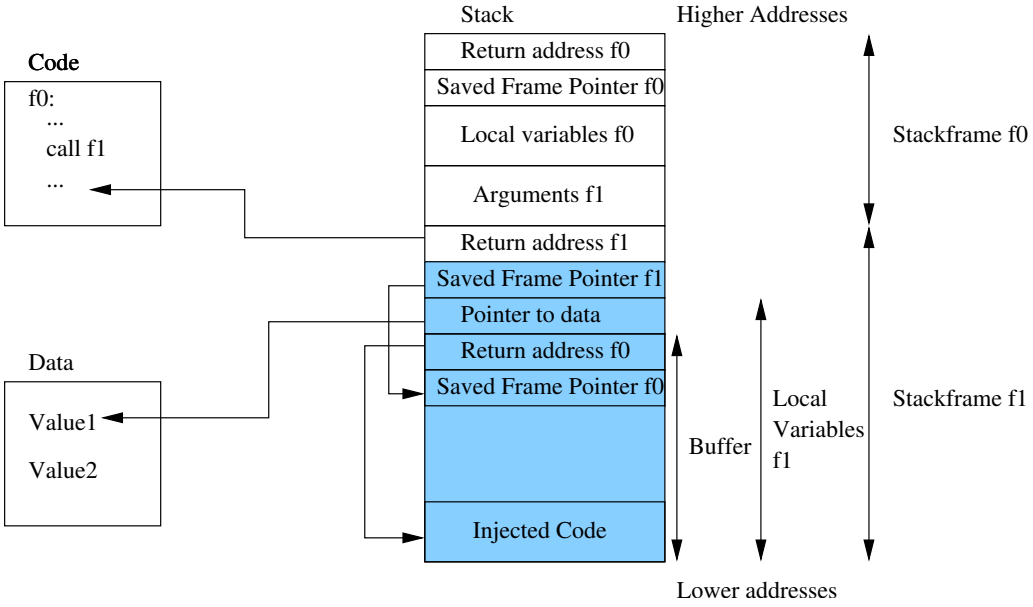


Figure 3: Stack-based buffer overflow overwriting frame pointer

frame pointer to still be able to gain control over the execution-flow of the program: the saved frame pointer would be set to point to a different location instead of to the saved frame pointer of the previous stackframe [68]. When the function ends, the frame pointer register will be moved to the stack pointer

register, effectively freeing the stack of local variables. Subsequently the old frame pointer register of f0 will be restored into the frame pointer register by popping the saved frame pointer off the stack. Finally the function will return by popping the return address in the instruction pointer register. In our example attackers have changed the saved frame pointer to point to a value they control instead of the frame pointer for stackframe f0. When the function f1 returns, the new saved frame pointer for f0 will be stored into the register which points to attacker-controlled memory. When the frame pointer register is used to 'free' the stack during f0's function epilogue, the program will read the attacker-specified saved frame pointer and will transfer control to the attacker's return address when returning.

### Indirect pointer overwriting

If attackers, for some reason, can't overwrite the return address or frame pointer directly through an overflow (some countermeasures prevent this), they can use a different technique illustrated in figure 4 called indirect pointer overwriting [18] which might still allow them to gain control over the execution-flow.

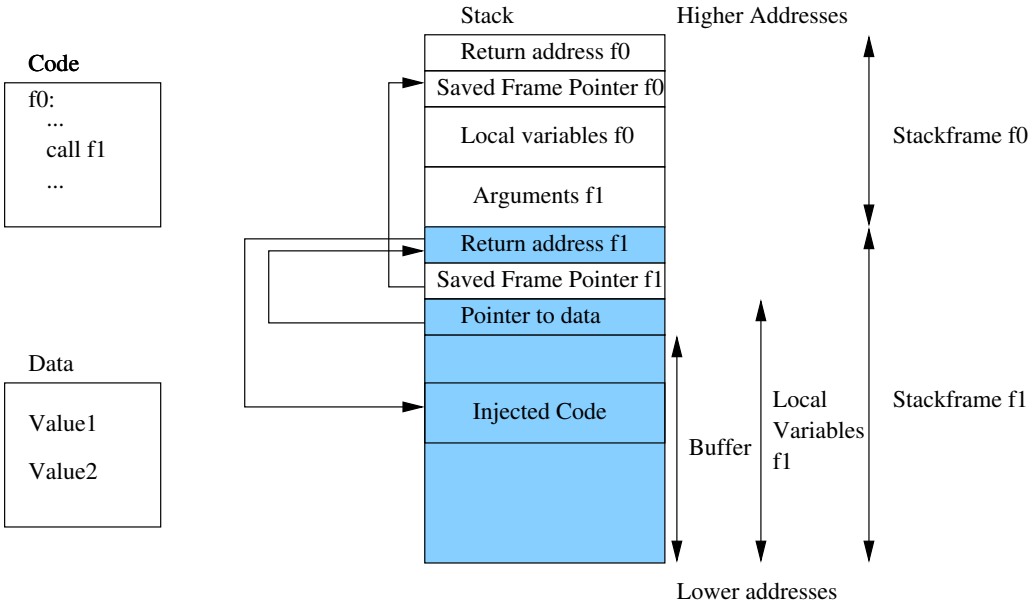


Figure 4: Stack-based buffer overflow using indirect pointer overwriting

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 they 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 later executed could be interesting for an attacker to overwrite.

### 2.1.3 Countermeasures and solutions for stack-based buffer overflows

This section contains a short summary of the countermeasures for stack-based buffer overflows. These are described more thoroughly in section 3.

The safe languages described in this document offer solutions for the buffer overflow problem by combining static checks and run-time checking. Statically they attempt to decide if array indexes or pointer arithmetic operations are safe. If they can't, they will either force the programmer to modify his code so that the compiler can prove them safe or will add run-time bounds checks to ensure the safety of these operations. Control-C [32, 69] on the other hand provides array safety by restricting the operations that can be performed on arrays and pointers and statically ensuring that these restrictions are abided by.

Static analyzers try to detect possible buffer overflows at development time by analyzing the code of the application: if an operation on a buffer or pointer is not considered safe, it will be reported as a place that might need extra verification by a programmer to ensure that the performed operations are safe. Some analyzers require assistance through annotation when analyzing a program, while others try to derive the assumptions the programmer has made themselves.

Dynamic analyzers instrument the application with checks and run-time information that will attempt to find buffer overflows when the program is run during testing phases. This added information can be used at run-time to verify that a specific array indexation or pointer operation would not write past the end of an array.

Sandboxes attempt to limit the amount of damage that attackers could do if they are able to exploit a buffer overflow vulnerability and are able to inject and execute their own code. Many of these enforce 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. Others attempt to do the same for file accesses by changing the program's root directory (chroot) and mirroring files under this directory structure that the program can access. Anomaly detectors have a similar approach, but instead of terminating the program when an unusual system call is performed they attempt to detect intrusions using this technique by recording the attempt.

Bounds checking compilers 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. Other compiler patches try to protect important information on the stack like the return address from overwriting. They generally do this by saving the return address to the heap or by placing a value on the stack before the return address an verifying it before returning from the function. PointGuard encrypt pointers so they are unusable when stored in memory but are decrypted before they are used in an operation.

Several hardware and operating system modifications also try to protect against buffer overflows: some mark the stack or data segments non-executable to prevent an attacker from executing injected code, others use the same techniques as some compiler patches by saving the return address in a different location and using that address when returning from a function. There are also hardware modifications that try to make injecting code hard by encrypting the

instruction set, making it hard for the attacker to guess the correct machine code representation of an instruction when injecting code. Memory randomizing countermeasures take advantage of the fact that attackers must also know exactly where their code is injected to be able to change the return address to point to their injected code. By randomizing the address on which the stack starts and randomizing where other parts of memory start, these countermeasures make it harder for attackers to guess the location of their injected code.

Some libraries offer the same kind of memory randomization as the operating system modifications. There are also libraries that will attempt to verify the integrity of the stack frames when performing string manipulations.

## 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 overflown 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

#### 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 overflown [23]. Function pointers are not always available though, so other ways of exploiting heap-based overflows are by overwriting a heap-allocated object's virtual function pointer [97] 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.

#### 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 [4, 10, 63, 117] can be a more general way of attempting to exploit a heap-based overflow.

The countermeasures we will be examining are based on a specific implementation of a dynamic memory allocator called *dlmalloc* [75]. We will describe this allocator in short and will describe how an attacker can manipulate the application into overwriting arbitrary memory locations by overwriting the allocator's memory management information.

The *dlmalloc* library is a run-time memory allocator that divides the heap memory at its disposal into contiguous chunks, that change size as the various allocation and free routines are called. An invariant is that a free chunk never borders another free chunk when one of these routines has completed: if two free chunks had bordered, they would have been coalesced into a larger free chunk.

These free chunks are kept in a doubly linked list of free chunks, sorted by size. When the memory allocator at a later time requests a chunk of the same size as one of these free chunks, the first chunk in the list will be removed from the list and will be available for use in the program (i.e. it will turn into an allocated chunk).

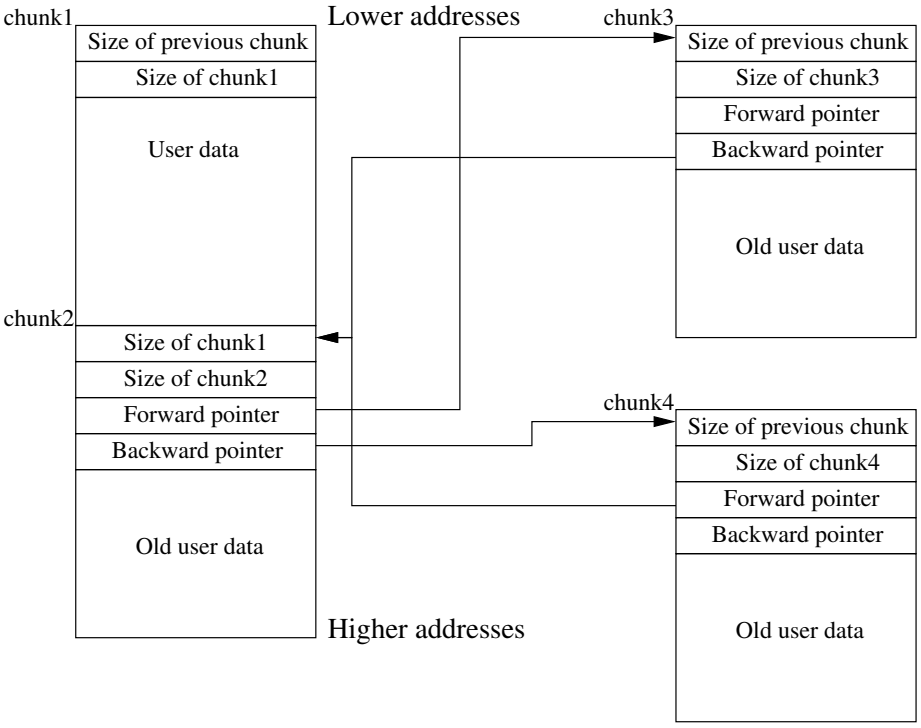


Figure 5: Heap containing used and free chunks

All memory management information (including this list of free chunks) is stored in-band (i.e. the information is stored in the chunks: when a chunk is freed the memory normally allocated for data is used to store a forward and backward pointer). Figure 5 illustrates what a heap of used and unused chunks could look like. $Chunk1$ is an allocated chunk containing information about the size of the chunk stored before it and its own size[1]. The rest of the chunk is available for the program to write data in. $Chunk2$[2] shows a free chunk that is located in a doubly linked list together with *chunk*3 and *chunk*4. $Chunk3$ is the first chunk in the chain: its backward pointer points to *chunk*2 and its forward pointer points to a previous chunk in the list. $Chunk2$ is the next chunk, with

[1]The size of allocated chunks is always a multiple of eight, so the three least significant bits of the size field are used for management information: a bit to indicate if the previous chunk is in use or not and one to indicate if the memory is mapped or not. The last bit is currently unused. The "previous chunk in use"-bit can be modified by an attacker to force coalescing of chunks. How this coalescing can be abused is explained later.

[2]The representation of *chunk*2 is not entirely correct: if *chunk*1 is in use, it will be used to store 'user data' for *chunk*1 and not the size of *chunk*1. We have elected to represent it this way as this detail is not entirely relevant to the discussion.

its forward pointer pointing to *chunk*3 and its backward pointer pointing to *chunk*4. *Chunk*4 is the last chunk in our example: its backward pointer points to a next chunk in the list and its forward pointer points to *chunk*2.

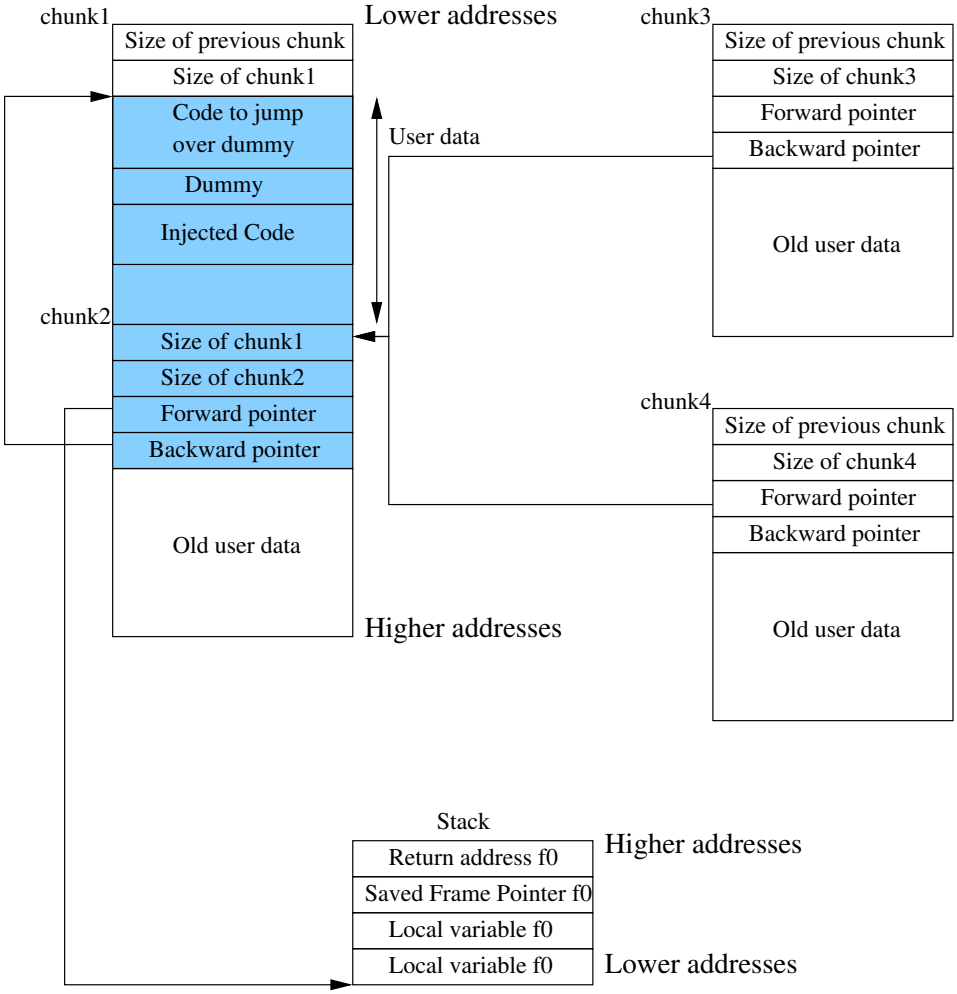


Figure 6: Heap-based buffer overflow

Figure 6 shows what could happen if an array that is located in *chunk*1 is overflown: an attacker has overwritten the management information of *chunk*2. The size fields are left unchanged in this case (although these could be modified if needed). The forward pointer has been changed to point to 12 bytes before the return address and the backward pointer has been changed to point to code that will jump over the next few bytes. When *chunk*1 is subsequently freed, it will be coalesced together with chunk2 into a larger chunk. As *chunk*2 will no longer be a separate chunk after the coalescing it must first be removed from the list of free chunks. The unlink macro takes care of this: internally a free chunk is represented by struct containing the following unsigned long integer fields (in this order): $prev\_size$, $size$, $fd$ and $bk$. A chunk is unlinked as follows:

```
chunk2->fd->bk = chunk2->bk
chunk2->bk->fd = chunk2->fd
```

Which is the same as (based on the struct):

```
*(chunk2->fd+12) = chunk2->bk
*(chunk2->bk+8) =  chunk2->fd
```

So the value of the memory location located 12 bytes after the location that $fd$ points to will be overwritten with the value of $bk$. And the value of the memory location 8 bytes after the location that $bk$ points to will be overwritten with the value of $fd$. So in the example in figure 6 the return address would be overwritten with a pointer to code that will jump over the place where $fd$ will be stored and will execute code that the attacker injected. As with the indirect pointer overwrite this technique can be used to overwrite arbitrary memory locations.

### 2.2.3 Countermeasures and solutions for heap-based buffer overflows

The same techniques that safe languages use to protect the program from stack-based buffer overflows can also be used to protect against heap-based buffer overflows. The same is true for static analysis, dynamic analysis, sandboxing, and bounds checking compilers. For the other compiler modifications that were discussed in 2.1.3 only the one encrypting pointers will also protect against heap-based buffer overflows as all pointers are encrypted, meaning the attacker can not reliably overwrite a code pointer.

The operating system and hardware modifications that offer encryption of instruction sets and randomization of memory areas also offer protection against this kind of overflow as they make it harder for the attacker to execute injected code.

The library modifications are the main place where countermeasures that try to protect against exploitation of *dlmalloc* are allocated. Mainly because the dynamic memory allocator is implemented as a library so calls to it are most easily intercepted there. Some of the countermeasures here offer a similar solution to the one used by the compiler patches to protect return addresses, they place a value after a specific chunk and verify this before exiting a string or memory copying function. Robertson et al. [99] protect the chunks by adding a checksum value in its header that makes sure that the memory management information has been left unchanged. Before freeing a chunk the checksum is verified. Other types of library modifications to protect against heap-based buffer overflows are also described in the 3.8.2.

## 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 as the value could have been changed at any place in the program.

However, double free vulnerabilities are a specific version of the dangling pointer reference problem that could lead to exploitation. 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 [33].

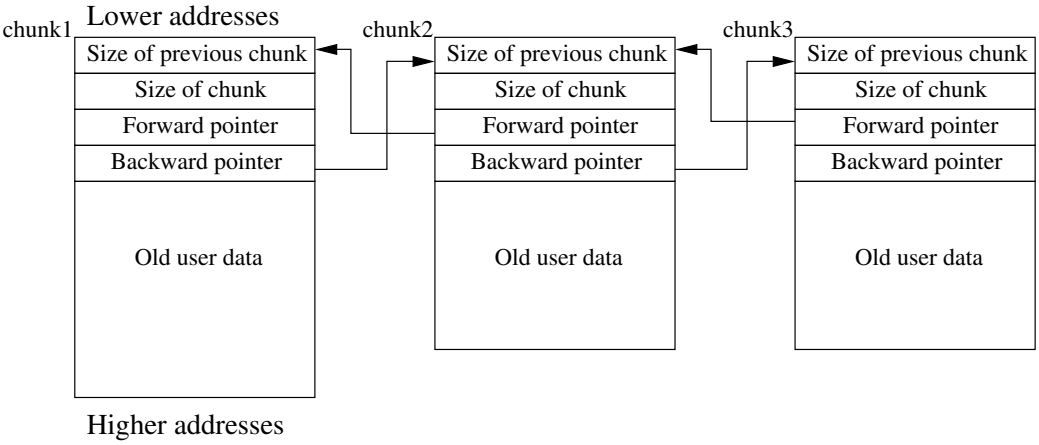


Figure 7: List of free chunks

Figure 7 is an example of what the list of free chunks of memory might look like when using the *dlmalloc* memory allocator. *Chunk*1 is bigger than the *chunk*2 and *chunk*3, meaning that *chunk*2 is the first chunk in the list of free chunks of its size. When a new chunk of the same size as *chunk*2 is freed it is placed at the beginning of this list of chunks of the same size by modifying the backward pointer of *chunk*1 and the forward pointer of *chunk*2.

When a chunk is freed twice it will overwrite the forward and backward pointers and could allow an attacker to overwrite arbitrary memory locations at some later point in the program.

### 2.3.2 Exploitation

As mentioned in the previous section: if a chunk (*chunk*4) of the same size as *chunk*2 is freed it will be placed before *chunk*2 in the list. The following code snippet does this:

```
BK = front_of_list_of_same_size_chunks
FD = BK->FD
chunk4->bk = BK
chunk4->fd = FD
FD->bk = BK->fd = chunk4
```

The backward pointer of *chunk*4 is set to point to *chunk*2, the forward pointer of this backward pointer (i.e. chunk2−>fd = chunk1) will be set as the forward pointer for *chunk*4. The backward pointer of the forward pointer (i.e. chunk1−>bk) will be set to *chunk*4 and the forward pointer of the backward

pointer (chunk2−>fd) will be set to *chunk*4. Figure 8 illustrates this (*chunk*3 is not shown in this figure due to space restraints).

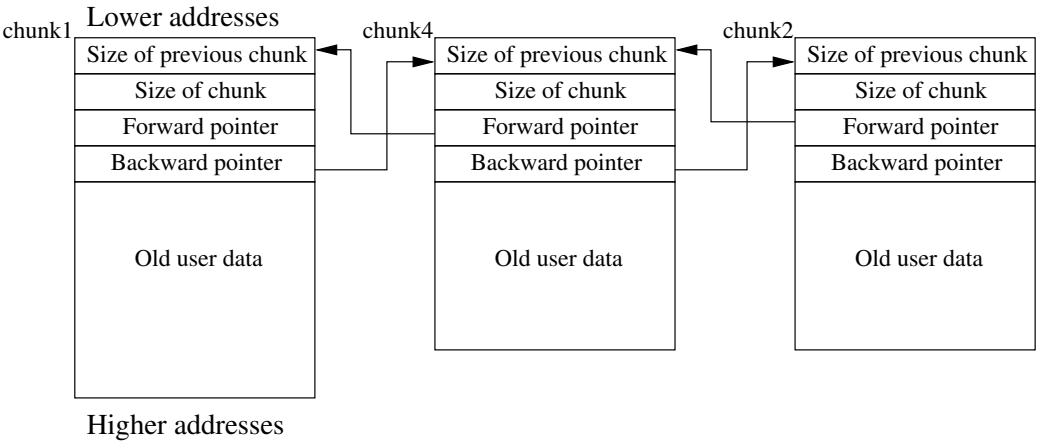


Figure 8: *Chunk*4 added to the list of free chunks (*chunk*3 not shown)

If chunk2 would be freed twice the following would happen (substitutions made on the code listed above):

```
BK = chunk2
FD = chunk2->fd
chunk2->bk = chunk2
chunk2->fd = chunk2->fd
chunk2->fd->bk = chunk2->fd = chunk2
```

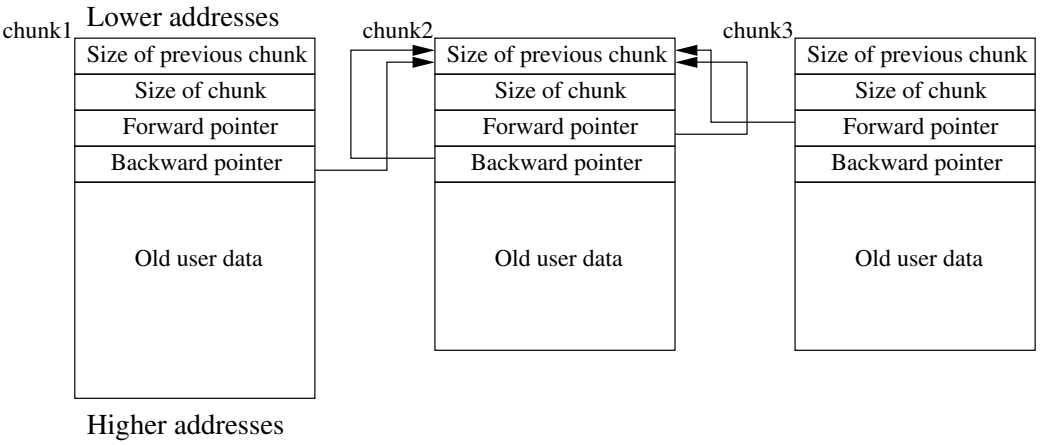


Figure 9: List of free chunks with chunk2 freed twice

The forward and backward pointers of *chunk*2 both point to itself. Figure 9 illustrates what the list of free chunks looks like after a second free of *chunk*2.

If the program subsequently requests a chunk of the same size as *chunk*2 then *chunk*2 will first be unlinked from the list of free chunks:

```
chunk2->fd->bk = chunk2->bk
chunk2->bk->fd = chunk2->fd
```

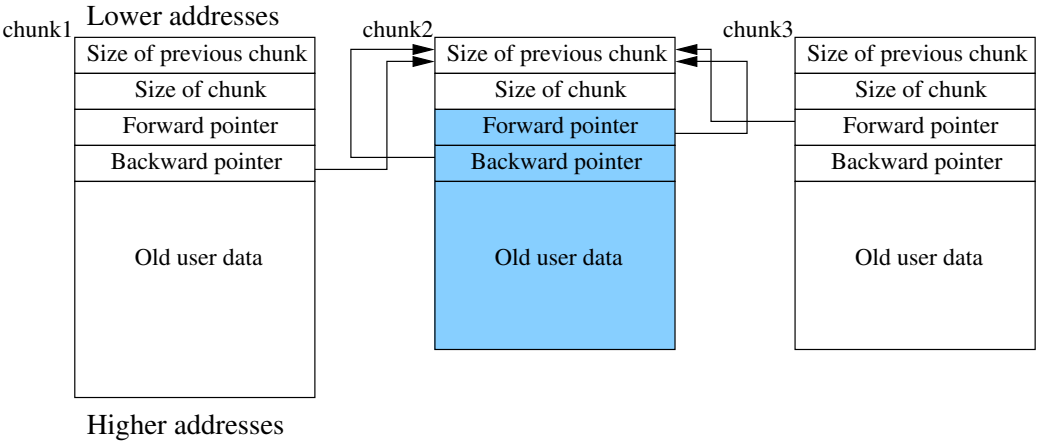


Figure 10: *Chunk*2 reallocated as used chunk

But since both chunk2->fd and chunk2->bk point to *chunk*2, it will again point to itself and will not really be unlinked. However the allocator assumes it has and the program is now free to use the user data part the chunk for its own use. Figure 10 illustrates where the program can now write.

Attackers can now use the same technique as was used in section 2.2.2 to exploit the heap-based overflow (figure 11): they set the forward pointer to point 12 bytes before the return address and change the value of the backward pointer to point to code that will jump over the bytes that will be overwritten. When the program tries to allocate a chunk of the same size again, it will again try to unlink *chunk*2 which will overwrite the return address with the value of $chunk2's$ backward pointer.

### 2.3.3 Countermeasures and solutions for dangling pointer references

The safe languages attempt to eliminate dangling pointer references by removing explicit control over memory allocations. In many of the safe languages the free operation no longer exists. To be able to deallocate memory that is no longer being used, region based memory management or garbage collection is used. This effectively prevents dangling pointer references.

Some bounds checking compilers will also protect against dangling pointer references by recording an extra temporal state for pointers, denoting if they are still active or not. If a pointer is freed the pointer is marked as being deallocated and any further dereferencing will generate an error.

Again the sandboxing, encryption, randomization and non-executable memory countermeasures can offer some protection against these kind of vulnerabilities because they make it harder for an attacker to execute injected code.

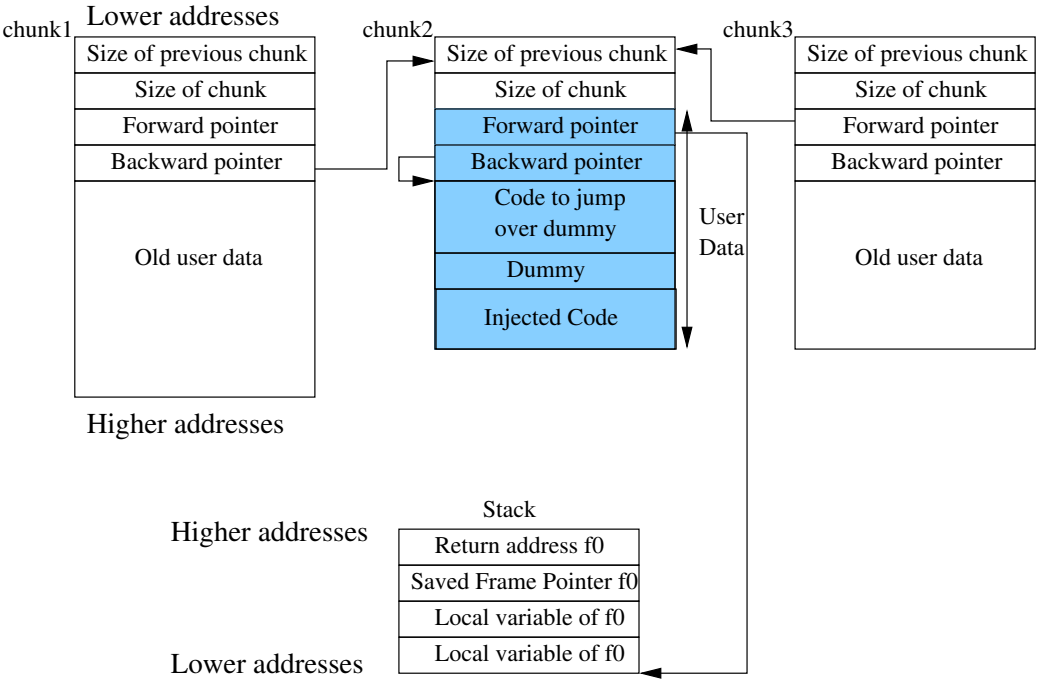


Figure 11: Overwriting the return address using a double free

Invalid use of dangling pointer references can also be detected by some static and dynamic analyzers.

## 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 (e.g. 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.

### 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 [1].

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 [105]. 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 8 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 effectively be used. 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 + 1 byte with another integer, then return address + 2 bytes and finally return address + 3 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 so 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 4 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). Figure 12 illustrates how an attacker would write 0x50403020 in an address using this technique.

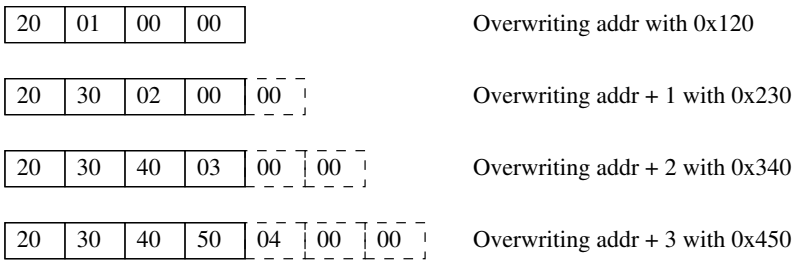


Figure 12: Overwriting a value 1 byte at a time using %n specifiers

### 2.4.3 Countermeasures and solutions for format string vulnerabilities

Some safe languages try to prevent format string vulnerabilities by allowing the format function to determine the type of its arguments, and comparing the type that was given as an argument to the function to the type of argument that the format specifier expects.

Static analyzers try to detect format string vulnerabilities by checking that the format string argument to a format function is either a literal string or by

doing taint analysis. Taint analysis marks all user input as tainted and will report an error when a variable that is expected to be untainted is derived from a tainted value.

Some library wrappers for format functions exist to try and prevent the exploitation of format string vulnerabilities. FormatGuard attempts to detect invalid format strings by counting the arguments the format specifier expects on the stack and comparing them to the amount of arguments that were really passed, if less arguments are passed than expected an error has occurred. Libformat checks if the format string that was supplied to the format function does not contain any %n specifiers if the string is located in writable memory.

## 2.5 Integer errors

Integer errors [16, 140] are not exploitable vulnerabilities by themselves, but exploitation of these errors could lead to a situation where the program becomes vulnerable to one of the previously described vulnerabilities. Two kinds of integer errors that can lead to exploitable vulnerabilities exist: integer overflows and integer signedness errors. An integer overflow occurs when an integer grows larger than the value that it can hold. The ISO C99 standard [2] mandates that unsigned integers that overflow must have a modulo of MAXINT+1 performed on them and the new value must be stored. This can cause a program that does not expect this to fail or become vulnerable: if used in conjunction with memory allocation, too little memory might be allocated causing a possible heap overflow. Nonetheless integer overflows don't usually lead to an exploitable condition.

Integer signedness errors on the other hand are more subtle: when the programmer defines an integer, it is assumed to be a signed integer, unless explicitly declared unsigned. When the programmer later passes this integer as an argument to a function expecting an unsigned value, an implicit cast will occur. This can lead to a situation where a negative argument passes a maximum size test but is used as a large unsigned value afterwards, possibly causing a buffer or heap overflow if used in conjunction with a copy operation (e.g. *memcpy*[3] expects an unsigned integer as size argument and when passed a negative signed integer, it will assume this is a large unsigned value).

### 2.5.1 Countermeasures and solutions for integer errors

As integer errors are not exploitable by themselves but can lead to a state in which a buffer overflow is possible the countermeasures and solutions to buffer overflows can be applied here too so many of those implicitly protect against integer errors too. However some static analysis tools attempt to detect integer errors by using taint analysis, if a variable that is derived from user input is used as an array index, a possible error is reported.

[3] *memcpy* is the standard C library function that is used to copy memory from one location to another

# 3 Solutions and countermeasures

This section examines the details of the solutions and countermeasures that have been proposed to combat the broad range of vulnerabilities described in the previous section. For the purpose of this document we define a solution as a complete solution to the problem, without ways of bypassing it. A countermeasure is defined as an attempt to make it harder to exploit a vulnerability. We have divided the different solutions and countermeasures into several categories depending on how they attempt to solve the problem or try to thwart an attacker.

## 3.1 Safe language solutions

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 general techniques that are employed by these safe dialects will be shortly described here, the more specific implementations are described later. In an attempt to prevent dangling pointer references, memory management is handled differently by these safe languages: the programmer is not given explicit control over deallocation anymore i.e. 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:

**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".

**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.

To prevent the other implementation errors that we described in section 2 several techniques are usually combined: firstly static analysis (see section 3.2 for a description) is performed to determine if a specific construct can be proven safe. If the construct can't be prove safe, then generally run-time checks are added to prevent these errors at run-time (e.g. if use of a specific array can't statically be determined to be safe then code that does run-time bounds checking might be added).

### Jim et al. (Cyclone)

Cyclone [61] was designed to prevent all the implementation errors described in this document. It does a static analysis of the C source code and will insert

dynamic checks in places that it can not determine to be safe. If the compiler is unable to determine if a program is safe, even after inserting run-time checks, it will refuse to compile it. In cases where a safe program is rejected, annotations that provide more information might be required to allow the static analyzer to correctly verify its safety.

The way pointers are treated has been changed too: whenever a pointer is dereferenced, a NULL check is performed on it to prevent programs from attempting to deference a NULL pointer as this might cause a segmentation violation. However these type of checks could slow down the program, so a new type of pointer is introduced: a never-NULL pointer (indicated by @), which is guaranteed to never contain a NULL value and thus NULL checks can be avoided. To prevent buffer overflows, pointer arithmetic on normal (*) and never-NULL (@) pointers is prohibited. To accommodate pointer arithmetic, a new kind of pointer is introduced (indicated by ?). This pointer contains extra bounds information that allows Cyclone to insert bound checks at relevant points in the program. The use of uninitialized pointers is also prevented, by performing static analysis to determine if a pointer has been initialized or not.

Dereferencing of dangling pointers is prevented using region-based memory management [55]. There are 3 types of memory regions in Cyclone:

- The heap consists of one large region that lives forever. There is no free operation defined to support freeing of heap-allocated objects.
- Stack regions have the same lifetime as a stackframe in C: a region is created that contains the function's arguments, return address and local variables. After returning from the function, the region is deallocated entirely.
- Dynamic regions are regions in which objects can be allocated independent of the heap or stack. All objects in this region are deallocated as a whole when the region is deallocated. They can no longer be freed individually.

Finally, format string vulnerabilities are prevented through the introduction of tagged unions: these allow a function to determine the type of its arguments. As such, format string functions can compare a format specifier with the type of a given argument, and can abort if an inconsistency is detected. Specifying tagged unions for every format string function argument can be inconvenient for programmers, so Cyclone performs 'automatic tag injection' which makes sure that the compiler inserts the correct tag for every argument at compile-time. The specific format string functions are implemented to check for consistency between tags and format specifiers.

### Necula et al. (CCured)

CCured [85] also uses static analysis on the source code that it is supposed to protect and, just like Cyclone, adds run-time safety checks in places that it can not statically determine to be safe. The main difference with Cyclone lies in the way pointers are handled and how memory is managed. While Cyclone allows a programmer to control which type of pointers are used in a specific place, CCured does not. Though CCured has safer pointers too, the programmer has no control over which type of pointer is used in which situation: static analysis

will decide what type of pointer is used. A pointer is safe if no casts or arithmetic operations are performed on it and the only checking that must be done on these types of pointers is a null-pointer check. A pointer is marked as sequenced if only arithmetic operations are performed on it and only null-pointer and bounds checking must be done at run-time. Finally dynamic pointers are pointers who's type can not be determined statically. These dynamic pointers require more extensive checking to ensure correct use, besides null-pointer checks and bounds checking, run-time tag manipulation is required. The main advantage of CCured's approach to pointer usage compared to Cyclone's is that legacy code (written in C) must not be modified extensively to be able to compile with CCured.

To prevent dangling pointer references CCured uses a garbage collector to manage its memory. Explicit deallocation of memory is ignored and a conservative garbage collector is used to reclaim memory.

The original CCured had some compatibility problems with existing C-code for which solutions were described in [22]: to avoid treating downcasts and upcasts as bad, CCured uses physical subtyping to determine if an upcast is valid: an object $T$ is a physical subtype of $T'$ if the concatenation of $T'$ and some $T$" are equal to $T$. To allow for downcasts, a new type of pointer called a run-time type information pointer is introduced. This run-time type information is used to determine if a downcast is valid: for a downcast of $T'$ to $T$, $T$ must be a physical subtype of $T'$. Furthermore, to provide compatibility with code that has not been compiled with CCured, a compatible way of representing arrays and pointers was introduced (for those objects that need to be passed to non-CCured-compiled code): the metadata that is kept for every object in CCured is separated from the object and stored in a similar structure, so the object can be passed back and forth to unprotected code with minimal effort.

#### Deline and Fähndrich (Vault)

Vault [82, 74], like Cyclone, uses region-based memory management. However it has a very different way of implementing this memory management mechanism. Vault allows a programmer to describe resource management protocols [31] that can be enforced statically by the compiler. It does this by offering an extension to the normal type system: a type has a type guard, which determines if access to the object is permitted. A guarded object's key must be in the held-key set (a global set of keys that are currently available i.e. held) before that object can be accessed. Function types also have pre- and postconditions denoting which keys must be held before that function can be called and which keys are set when the function returns. Vault implements its region-based memory management using these keys: when a region is created a key for that region is held and all objects in that region are type guarded by that key. When the region is subsequently deleted the key is removed from the held key-set and any access to an object in the region will cause the compiler to return an error.

Vault differs from C in more areas to make it a safe language: variables are always initialized to either a programmer specified or a default value, preventing the use of uninitialized values. Normal * pointers can not contain a NULL value, they must always point to a valid memory location. To allow the programmer to express pointers that can contain NULL values ? pointers are provided. And finally, type safety is provided by prohibiting arbitrary type casts: only type

casts between values of the following types are allowed: byte, char, short, int, long, long long and string (which is a separate type in Vault as opposed to a char * in C).

### Kowshik, Dhurjati, Adve and Lattner (Control-C)

The aim of the approach taken by Control-C [32, 69] is to ensure memory safety in untrusted C programs without run-time checks, annotations or garbage collection. It tries to statically guarantee memory safety, but to achieve this only a subset of the C language is supported and specific assumptions are made about the system. It is specifically designed for use in real-time control systems and runs on the Low Level Virtual Machine (LLVM) system, where all memory and register operations are type safe.

To work efficiently some assumptions are made about the run-time system:

- Certain run-time errors are considered to be safe (i.e. the system will kill the untrusted code generating the error).
- If the stack or heap grow beyond the available space a safe run-time error is generated.
- Existence of a reserved address range that, on any access, generates a safe run-time error.
- Certain standard library functions and system calls are assumed to be safe.

The following restrictions must also be abided by:

For type safety:

- All variables, assignments and expressions must be strongly typed.
- Pointers may not be cast to other types.
- Unions may only contain types that can be cast to one another.

The compiler can easily check type safety within the LLVM system, as only explicit cast instructions can break type safety, so only these must be checked by the compiler.

For pointer safety:

- Local pointers must be initialized before being dereferenced. To detect uninitialized local pointers, data-flow analysis is used. However this is not as easy for global pointers and pointers in dynamically allocated structures. To ensure that these are initialized, run-time code is added to initialize them.
- Any data type should be smaller than the size of the reserved address range. This follows from the second run-time system assumption.
- The address of a stack location can not be stored in a heap allocated object or global variable nor can it be returned by a function. This is enforced by doing data structure analysis, a directed graph of all memory objects in a function is constructed together with their location and information

about where they point to. The graph also contains information about the function's parameters, return values and reachable objects passed by callers or returned by callees. By traversing the graph it is possible to determine if a stack allocated variable is reachable from where it is being pointed to.

For array safety:

- The value of the index used in an array access must lie inside the bounds of the array.
- Dynamically allocated arrays must have a positive size.
- If an array is accessed from inside a loop then the bounds of the loop must be an affine transformation of the array's size and the loop index variables. The index used in the array access must be an affine transformation of the loop index variables or of the size of the array. If the index value depends on a symbolic variable, then the memory accessed must be independent of the variable.
- For an array access outside the loop, the index value must be an affine transformation of the size of the array.

The first restriction ensures array safety and follows from the conformance of the program with the three subsequent restrictions.

Finally, dangling pointer reference problems are prevented using region-based memory management. However, explicit deallocation of memory is allowed, which would reintroduce the problem. However, these dangling references are only a problem when used to violate type safety. To prevent that, variables of the same type are placed in a region which contains variables of a homogeneous type. To implement this an algorithm called Automated Pool Allocation is used where a different pool is allocated for every type when the first variable of that type is allocated.

## 3.2 Static analyzers

Static analyzers are generally used during the implementation or audit phases of an application, although they can also be used by compilers to decide if a specific run-time check is necessary or not (as seen in the previous section). They do not offer any protection by themselves but are able to point out what code might be vulnerable. They operate by examining the source code of a particular application. They can be as simple as just searching the code for specific known vulnerable functions (strcpy, gets, ...), or as complicated as building a complete model of the running application.

However, determining statically, for any possible input program, if a program will contain an overflow or not is an undecidable problem: it is trivial to reduce this problem to the halting problem. So all analyzers will contain a number of false positives (the amount of correct code that is incorrectly reported as being vulnerable), false negatives (vulnerable code that is not reported as such) or both. The main criteria for determining the effectiveness of a source code analyzer is the false positive to false negative ratio, and how well they scale to larger, real-world software systems. An important difference can also be

made between sound and unsound static analyzers: sound analyzers will find all possible overflows, usually at the expense of generating more false positives and being less scalable while unsound analyzers try to find the right balance between false positives and false negatives and being able to analyze larger systems.

For our purposes we will divide these analyzers into two groups: annotated, which place much of the analysis burden on programmers by forcing them to explicitly document their assumptions and non-annotated that try to infer what the programmer implied from the source code.

### 3.2.1 Annotated source code analyzers

Annotated source code analyzers expect the programmer to help the analyzer in determining which assumptions specific code is making. E.g. a programmer could expect a specific pointer not to be equal to NULL and would then annotate it as such. A source code analyzer would then attempt to certify that this pointer can indeed not be NULL at that point in the code. Annotations make the life of an analyzer considerably easier as it has less guesswork to do about what the programmer intended but can rely on the programmer to tell it what they are expecting and can focus on attempting to assert these expectations. The advantage of this technique is that it allows analyzers to quickly and easily search the source code for specific places where assumptions might not be met.

#### Larochelle and Evans (Splint)

Splint [40, 72]. is a lightweight static analysis tool, that analyzes the program source code based on annotations that describe assumptions made about specific objects (i.e. specific value, lifetime, ...) or pre- and postconditions that must be met for a specific function.

By providing some built-in annotations for specific 'vulnerable' functions (e.g. for strcpy: */∗ @requires maxSet(s1) <= maxRead(s2) @∗/* i.e. the size of $s1$ (destination) is larger than the amount of data read from $s2$ (source)) Splint can be used with a minimum of effort by a programmer to verify source code for commonly encountered implementation errors such as the ones described in section 2 of this document.

Format string vulnerabilities are detected using taint analysis. All user input is considered tainted and an error is reported if a tainted variable is used where an untainted one is expected. The definitions of the format functions are then changed to expect an untainted format strings as arguments.

#### Dor, Rodeh and Sagiv (CSSV)

CSSV [34, 35] is a static source code analyzer designed to detect all buffer overflows in C. It will report all errors at the expense of reporting some false positives. CSSV expects a programmer to provide a contract that describes the preconditions, the postconditions and the side-effects of a function. It can however also derive such contracts by itself. Analysis is performed by first inlining the contracts by means of asserts and then translating the source code to CoreC [139]: a semantics-preserving subset of the C language (i.e. a simplified version of C to which all possible C programs can be transformed without changing their semantics). This allows for simpler static analyzers and source-to-source

translators as only a subset of the language C must be supported instead of the complete language.

After translation, pointer analysis is performed to detect which pointers point to the same base address. These results are used to transform the program to an integer program, after which integer analysis [24] is performed. This analysis will indicate which inequalities between variables are guaranteed to hold at every control point. During this analysis all asserts added by the inlining of the contracts are verified. If an assert fails, the existence of a possible error is announced. The number of false positives that CSSV will return depends on how accurately the contracts are defined.

#### Shankar, Talwar, Foster and Wagner (extended cqual)

Cqual is an extensible type qualifier framework [48]. The authors extend this tool to find format vulnerabilities [108]: all user controlled data is marked as tainted, as are all variables that are assigned a value that is derived from tainted data. If tainted data is ever used as an argument to a format string, an error is raised. Tainting is statically modeled by extending the C type system with new type qualifiers: besides const and volatile, tainted and untainted are added. To make it more easily backwards compatible, the tool comes with default annotations for the C library functions. To further relieve the programmer of the burden of having to specify type tainted or untainted for every variable, type inference is used: by default the return value from calls to functions like gets are tainted and every variable that is derived from one of these variables is marked as tainted.

When on of these variables is used where an untainted one is expected, an error together with a flowpath display is reported to allow the programmer to find where the error occurred. To use this technique to find format string vulnerabilities, format functions are defined to expect untainted variables as format string specifiers. For example, the function printf (**const char** ∗format, ...) would be defined as printf (**const** untainted **char** ∗format, ...).

#### Ashcraft and Engler (Metacompilation)

Ashcraft and Engler [5] suggest allowing programmers to write compiler extensions where they specify that the source code must adhere to specific rules. The compiler then statically checks if the source code adheres to these restrictions. Belief inference is used to attempt to find faults in the checker, i.e. to guess what the programmer believed at a particular point in the code, e.g. if a value is read from an untrusted source and sanitized, the programmer believes that it will be used for something sensitive, if the checker does not find such a call, it assumes the check missed a possibly dangerous operation and reports this.

They demonstrate their metacompilation technique with example specifications that search for the use of integers from untrusted sources (i.e. tainted variables) without checks in the Linux and BSD kernels. This use of tainted integers might cause integer errors and array bounds violations (when used as indices for arrays) which could lead to vulnerabilities that could be exploited by an attacker.

### 3.2.2 Non-annotated source code analyzers

These kind of source code analyzers do not require annotation about specific assumptions made by the programmer to try and determine if code is safe. They generally attempt to assert that code is safe by building a model of the execution environment at every step of the code (i.e. a sort of C interpreter) or by reducing the program to a simpler system in function of the buffers (e.g. a set of constraints or integer ranges). By following all the possible execution paths, they are normally able to certify if a specific assumption that is being made always holds. For example, if a program dereferences a pointer without first doing a NULL check, but they are able to certify that the pointer will never be NULL at that point of the code, then that code is safe, otherwise the analyzer must take some predetermined action (like notifying the programmer of specific problematic code).

#### Bush, Pincus and Sielaff (PREfix)

PREfix [19, 74] wishes to offer the same kind of functionality as Purify [56] (which we describe later in section 3.3) without the need for a run-time execution of the program that is being examined. It is a static source code analyzer meaning that it requires access to the underlying source code, while Purify only requires access to the object code. PREfix builds an execution model of the examined source code: it does a sequential trace of achievable execution paths and actions are simulated on a virtual machine. The results of simulating these execution paths are then used to build a model of each function. A bottom-up traversal of the call graph is performed, starting with the leaf functions, so that the model of the function can be used to generate a set of constraints on the callers of that function. This also means that sometimes values are unknown at the time a function is being examined and modeled. To prevent the analyzer from returning a large amount of false positives tests are delayed: a constraint is added in the model of the function that will be tested when the model is called.

Using this approach a full memory model of all possible execution-flows of an entire program is built and possible inconsistencies between expected and actual values are detected, while minimizing the amount of false positives that are reported.

#### Pincus (PREfast)

PREfast [74] is an adaptation of PREfix that was designed for speed. PREfix can take a long time to analyze a program and developers want to check their code for errors while developing it rather than running PREfix when a more mature program exists. As such PREfix and PREfast are complementary tools, PREfast can be run regularly during development to find simpler errors and PREfix can be scheduled to run at specific intervals to perform a more complete analysis. Because speed was an important objective when developing PREfast it is a more lightweight analysis tool that only performs local analysis. It parses the function's code and searches for code that is likely to cause an errors.

### Xie, Chou and Engler (ARCHER)

ARCHER [135] is designed to look for memory access errors. It transforms C code into an intermediate representation where side-effects are eliminated by introducing temporary variables, nested function calls are flattened and operators such as ? are converted to 'if' statements. Next a control-flow graph and an approximation of a call graph are built. The call graph is then analyzed bottom-up, with each function being analyzed for possible errors. The traversal module of ARCHER will explore each function's control-flow graph and will perform one of the following actions: if a condition is encountered, its expression is evaluated and if the evaluation can be determined statically, non-reachable control paths are eliminated; if a memory access is performed, an attempt is made to determine if the access is unsafe, if it is an error is returned; for all other statements, the statement is converted to a set of constraints that update the state. Because ARCHER does interprocedural analysis and actual parameters to a function are only known at the place where the function is called, a set of constraints is kept for parameters to a function. When the function is called, the parameters are compared to the constraints and if needed, an error is reported.

A major limitation to ARCHER is that it only operates on arrays and pointers in programs and does not understand string operations. Memory for which no size can be determined is also not checked.

### Simon and King

Simon and King [109] build on the work done by Dor et al. [34] that aims to improve some of their work's shortcomings: possible changes to a buffer are not tracked, only definite changes, e.g. **if** (rand()) q=s; **else** q=t; (with $s$ being a string ending with NULL at a different position than $t$ does and $q$ being a character pointer) then the actual value of $q$ can only be determined at runtime. To remedy this, possible values of $q$ must be tracked next to definite values. To perform analysis, the C program is reduced to a representation (called String C) that performs operations on buffers. Each pointer to a buffer is represented by a triplet: the buffer's starting point, the current position in the buffer and the buffer bounds. For valid accesses the following must hold: $start \leq current < bounds$. All information about a buffer is expressed in function of its triplet: i.e. the size of the buffer is $bounds - start$ and the position of the first zero in the buffer can be kept as an offset to start of the buffer. To simplify analysis, only the first zero is tracked as opposed to every zero, even though this might lead to some false positives.

### Wagner, Foster, Brewer, and Aiken (BOON)

Wagner et al. [127] propose an unsound static analysis technique for detecting buffer overflows. They model strings in C as integer ranges: each string is represented by two integers: the number of bytes allocated for the string and the number of bytes currently in use. A constraint language based on these ranges is defined and the specific string operations are modeled in this language (i.e. the constraints that must hold are specified). The program is subsequently parsed and a system of integer range constraints is generated for each statement in the input program. As each string is represented by two variables and all

string operations are modeled with regard to these values, the safe state is when $\forall\, s \mid len(s) < alloc(s)$.

To simplify analysis, the authors decided to use flow insensitive analysis (control-flow is ignored when generating constraints), this brings about some problems for functions like strcat which could be executed in a loop and have a state that relies on the previous execution. To alleviate this problem any call to strcat is flagged as a possible vulnerability, which will most likely generate some false positives. The current technique also has some limitations when analyzing pointer operations: it can not handle pointer aliasing correctly; ignores function pointers, doubly indirected pointers and unions.

After generation of constraints, the constraint system is solved by finding a minimal bounding box solution that encloses all of the possible execution paths. The places where the constraints might not hold are reported as possible overflows.

### Rugina and Rinard

Rugina and Rinard [100] describe a static analysis framework designed towards the detection of errors in parallelized programs. They focus on data races, array bounds violations and bitwidth analysis (allowing a problem to only use as many bits as required). First pointer and read-write set analysis is performed on the program to determine which memory regions are accessed by each instruction and each procedure.

Intraprocedural bounds analysis is used to derive bounds information for pointers and array indexes at every point in the program. Initially upper and lower bounds are generated for the variables at the start of the procedure, new bounds are then generated by symbolically executing instructions and generating bounds for every program point. These bounds are then used to build a constraint system over the upper and lower bounds. The constraint system is then solved using linear programming to arrive at a polynomial for the lower and upper bounds, i.e. bounds are expressed in terms of values rather than based on results of previous program steps.

Next intraprocedural region analysis is performed to determine what memory regions are accessed directly for reading or writing by a function. Then interprocedural region analysis is used to determine the regions accessed by an entire execution of a function (including the effects of calls to other functions). The information gathered from this phase allows the compiler to determine if the program might contain buffer overruns.

### Ganapathy et al.

Ganapathy et al. [50] describe a static analysis technique where each character buffer is modeled as four numbers: allocmax, allocmin, usedmax, usedmin denoting the maximum and minimum amount of bytes allocated and the maximum and minimum amount of bytes used. Subsequently a flow and context insensitive analysis is done to generate constraints on the program in terms of these variables. Specific string manipulation functions are modeled in terms of constraints that model the effect they have on a call.

As linear programming can only operate on finite values, any constraint variables that might assume an infinite value (when derived from user input

for example) are removed using taint analysis. Subsequently the constraint system is solved using linear programming: the best constraints are found for the usedmax and usedmin range and for the allocmax and allocmin range.

Finally, the values determined by solving the constraint system and the values gathered by the taint analysis are used to determine if a possible overflow could result (i.e. at some point, more bytes could be used than were allocated for a string).

#### Viega, Bloch, Kohno and McGraw (ITS4)

ITS4 [124] is a simple static analyzer that attempts to find unsafe C and C++ code. The main idea behind ITS4 was to design a scanner that is so lightweight that it should be feasible to implement it as an editor extension much like syntax highlighting. Unlike most static analyzers it will not parse the source code to do its analysis, instead ITS4 limits itself to doing lexical analysis and then matches the code to a vulnerability database that contains a list of unsafe functions or functions that are misused in many cases, each with a potential risk factor and a description that can be printed out when the possible vulnerability is encountered. When it has identified possibly vulnerable functions it will examine them further and determine if they are to be reported and with which level of severity. For example strcpy(dst, "\n"); will not be reported as a potential vulnerability because the source string is fixed and consequently the potential risk factor is extremely low.

#### Wheeler (Flawfinder)

Flawfinder [130] is very similar to ITS4. It also examines code at the lexical level and will report known vulnerable functions back to the programmer as a possible security vulnerability. It will prioritize the output in function of the potential risk that it poses.

#### "Secure Software, Inc" (RATS)

RATS [106] too is very similar to ITS4, however it can also examine Perl, PHP and Python code next to C and C++ code. It, just like the previous two countermeasures, will search for vulnerable functions and report them together with risk factor.

### 3.3 Dynamic analysis and testing

Dynamic analyzers have a different approach to analyzing programs, instead of trying to determine the presence of vulnerabilities at the source code level, they generally instrument the program to output extra information at run-time. If a possible vulnerability is found during the instrumented run, this is reported back to the tester. This allows for more exact checking than can be done statically, but might miss some errors because some execution paths might not have been executed during testing.

### Haugh and Bishop (STOBO)

STOBO [57] is a tool to dynamically detect possible buffer overflows in C programs. It instruments the source program and creates a version that when run, has the same behavior as the original program but outputs information on possible overflows. It will generate different types of warnings of possible buffer overflow vulnerabilities in the following situations:

1. When the source and destination are statically allocated (this is called type 0).
2. When the source is dynamically allocated and the destination is statically allocated (type 1).
3. When the destination is dynamically allocated (type 2).

It will of course only generate these type of warnings if the memory allocated for the destination is possibly smaller than that of the source and no adequate size checking is performed before copying one buffer to another.

As some size information is only available at run-time, the authors chose to implement their checking dynamically. The instrumentation keeps track of static buffers and their sizes by adding functions to record their existence right after their declaration in the source. To keep track of dynamically allocated buffers all calls to memory allocation and freeing functions are replaced with wrapper functions.

As the checking is done dynamically, some of the problems with generating high false positives and false negatives that static analyzers suffer from can be eliminated.

### Hastings and Joyce (Purify)

Purify [56] is a software testing tool designed to detect memory leaks and access errors, although its focus is not on security, some of the security vulnerabilities that we examine in this document are the result of memory access errors. Purify is designed as a software debugger that detects memory errors dynamically as opposed to detecting them statically. To accomplish this, it instruments a program's object code and inserts a function call that checks the memory access before every load or store instruction. This function checks and updates a memory state table, in which a 2 bit state code is kept for every byte allocated in the heap, stack, data or bss sections. These bits associate 3 states with the byte of memory: unallocated (unwritable and unreadable), allocated but uninitialized (writable but unreadable), allocated and initialized (writable and readable). When a write is done to uninitialized memory, it is marked as initialized. When a write is attempted to unallocated memory, a diagnostic message will be generated. To detect heap-based array overflows small areas marked as unallocated are placed before and after each chunk returned by malloc. If these bytes are either read from or written to, a memory access error will occur and the tester will be notified.

### Larson and Austin (MUSE extension)

Larson and Austin [73] suggest a dynamic analysis technique where the source code is instrumented to detect invalid array accesses. Unlike testing solutions,

they do not rely on the current value of entered data but instead keep a range of possible values for all input variables and if a possible error could occur (even if the actual input value does not cause it) an error is generated. For example: if an unsigned integer $x$ is a user-inputted variable that is used as an array index, at program start it will be in the range of $[0, MAXINT]$. If its value is checked with a condition, the range is modified to reflect the impact of that check (i.e. **if** (x <= 4) will result in the range of x being $[0, 4]$). This narrowing does not depend on the value supplied by the user: the whole range ($[0, 4]$) will be compared to the array size when it is used as an index.

To determine which integer type variables are input related (integers types are the only valid array index types), a form of tainting, as was described in the examination of extended cqual and metacompilation, is employed. These input variables are then assigned a range of possible values that they contain.

For strings, besides information about its size, the fact if it is known to contain a NULL or not will be recorded (e.g. when an strncpy occurs and no room is left for the NULL byte, the known_null state will be false). When a particular NULL terminating operation (like manually setting a NULL byte at the end of the string) occurs, the known_null flag will be set to true.

Other faults are also detected using these techniques: possibly negative values for memory copy or allocation operations will generate alerts. Integer overflows and underflows will also be detected by checking if the result can be stored in the destination variable or not.

### Ghosh, O'Connor and McGraw (FIST)

FIST [51, 52] attempts to discover vulnerabilities using software fault injection. Anomalous events are injected during the execution of the program and from the response to these events, it is determined if a violation of the security policy has occurred. FIST can inject a wide number of errors (changing boolean values, exploiting buffer overflows, ...) into the program and allows a programmer to monitor if it causes a violation. The most interesting of the injected faults for this document is the ability to generate a stack-based buffer overflow attack that will overwrite the return address of a function and will attempt to execute code that causes an event that is caught by FIST's monitoring component.

A major drawback however is that in order to do buffer overflow analysis using FIST, a programmer has to manually select which buffers are to be instrumented. To alleviate the potential burden this would cause for testers, the authors propose an automatic way of detecting buffers to test [51]. The source code is parsed and examined, and places that might be vulnerable to attacks are instrumented accordingly.

### Fink and Bishop (Property-Based Testing)

Property-based testing [46] is a methodology used to test if a specific property holds in a program and is used for program validation. The properties to test are specified formally and are used to check if the properties hold during test execution. The tests consist of a set of executions of a program using different input data each time to determine if the program functions correctly. If a test fails, a property does not hold or an unwanted side effect is generated. The program is first sliced, all code affecting the property to be tested is extracted

and a data-flow graph is produced. Afterwards the program is instrumented to keep track of when a specific part of the code was called and what the result was. Next tests are generated, either automatically or by the human tester. After test executions, coverage and correctness are evaluated: if specific parts that should have been tested were not executed, then coverage will be incomplete and more tests must be performed. If a property violation is recorded during a test then this can be reported as an error.

## 3.4 Sandboxing

Sandboxing is based on the "Principle of Least Privilege" [104, 102], where an application is only given as much privileges as it needs to be able to complete its task. This can be enforced in a number of ways:

- Policy enforcement: a clear policy is defined, some way or another, specifying what an application specifically can and can't do.
- Fault isolation: ensures that when a program or part of a program fails it will not cause the entire system to malfunction. The most common application of fault isolation is to load isolated code into its own hardware-enforced address space.

These kinds of countermeasures attempt to limit the amount of damage to a system that a component (i.e. a module introduced into an application or even a whole application) could do if an attacker is able to gain control over it.

### 3.4.1 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.

Several other suggestions of doing fault isolation are examined here. Although some 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).

#### Wahbe, Lucco, Anderson, and Graham (Software-Enforced Fault Isolation)

The concept of Software-based Fault Isolation (SFI) was first introduced by Wahbe et al. [128]. Techniques are provided to allow for fault isolation without the need for a separate address space. This avoids the need for context switches and allows for less expensive communication between modules at the cost of increased execution time. Wahbe et al. do the isolation in a straightforward manner: an application's address space is divided into segments such that the addresses in that segment all have the same unique most significant bit pattern.

To enforce this isolation, two techniques are proposed: segment matching and address sandboxing.

**Segment matching** adds a check before each jump or write instruction that accesses an address that can't be verified statically (indirect addressing) to be in the correct segment (referred to as an unsafe instruction). This check compares segment identifiers of the module and the instruction parameter and will cause an exception if they fail to match. This allows a programmer to find out where the unsafe memory access occurred.

**Address sandboxing** does things a little differently: instead of inserting a check before each unsafe instruction, every unsafe instruction's address parameter is set to the current module's segment identifier (i.e. the most significant bits are set to segment identifier's most significant bits). This reduces the execution-time overhead associated with enforcing SFI.

To allow writable memory sharing (read instructions are not modified) between fault domains, a technique called lazy pointer swizzling is proposed: the hardware page tables are modified to map the shared memory into every address space segment that requires access such that the shared region is mapped at the same offset in each segment.

These techniques, in their proposed implementation, can make it harder for an attacker to exploit the vulnerabilities described earlier as the options for memory addresses to overwrite are more limited because address modification instructions are either checked or modified before use at run-time. However they will probably not be able stop a buffer overflow where the adjacent memory space contains the return address and is overwritten. Once attackers are able to inject their own code they will again be able to gain complete control over the software system as the injected code will not contain any fault isolation enforcement. Nonetheless, it might still be possible to use these techniques as a possible countermeasure by enforcing the isolation at run-time by monitoring the execution and replacing the instructions with the checked or isolated ones.

#### Small (MiSFIT)

MiSFIT [111] is an extension of the SFI techniques described by Wahbe [128]. It sandboxes instructions to limit them to only read from and write to memory regions that the module is allowed to respectively read from and write to. However, an extra constraint is added: to prevent the code from modifying itself write instructions can not have an argument that points to its own code segment. To prevent the code from jumping to non-entry point code sections or data sections a table of valid functions must be provided for every module so that indirect function calls can only be used to jump to code-entry points. Global data is accessed through function calls, to which each module that requires access to the global data is given call permission in its table of valid functions.

The stack presents a problem for programs that must be completely isolated: it falls outside their memory region and contains some information, like return addresses that should not be changed by a module. Thus full control over the stack can not be given to a fault isolated module. MiSFIT solves part of the stack problem by assigning each module its own stack. To prevent it from

overwriting its own return address each call instruction is replaced with a jump to code that saves the return address to a stack outside the module's control space. Each return instruction is replaced with a jump to code that retrieves the saved return address and then jumps to it.

### 3.4.2 Policy enforcement

Compared to fault isolation, policy enforcement techniques do things a little differently: a specific policy of what the program can or can't do is written down and subsequently enforced. Generally this enforcement is done through a reference monitor where an application's access to specific resource (the term resource is used in the broadest sense: a system call, a file, a hardware device, ...) is regulated.

#### Erlingsson and Schneider (SASI)

SASI [38] merges enforcement of policies into the object code of an application: code to check if a security policy is adhered to is added to the application's object code. Security policies are defined by security automata. The input alphabet is defined by events that can be monitored (e.g. read, push, ret, ...), the translation relations describe the actual policy: events that are rejected by the automaton will be denied at run-time. SASI merges policies into applications in four phases: first the automata are inserted before each target instruction, then each transition is evaluated based on the instruction that follows, any irrelevant translations are deleted from the automaton and finally the automata are compiled into code that simulates the automata.

Because SASI merges its policy into existing application code, extra checks are needed to ensure that the application does not modify variables that are part of the security automata, nor circumvent the automaton simulation code altogether, nor modify its own code. These checks are implemented by verifying the object code to ascertain that these circumventions are not possible or, if they are, by modifying the object code to prevent them. However since the checks are added to the object code, it might still be possible for an attacker to use one of the aforementioned vulnerabilities to circumvent the protections added by SASI. As such, a run-time solution would be more appropriate in the current context.

#### Schneider

Schneider [103] examines a policy enforcement mechanism where the execution of a system is monitored (without modifying the original program or examining its source code beforehand) and when a potential violation of a predefined policy is recorded, the execution is terminated. Although these execution monitors encompass a broad range of possible systems: like kernels, firewalls, reference monitors, ...; our interest is related to the application of these techniques for programs that might be attacked through the previously described vulnerabilities. Execution monitors must be able to decide if a program violates a security policy based on individual executions. A set of these individual executions is called a safety property. Schneider points out that execution monitoring can only enforce security policies that are safety properties. He further defines a

formalism for defining security policies using security automata in the same way that is used in SASI. Although this kind of enforcement is primarily focused on execution of untrusted code, it can be useful in preventing attackers from injecting code that would violate the security policy that has been defined for the program.

### Evans and Twyman (Naccio)

Naccio [41] is a system architecture that allows safety policies to be defined for applications. Such policies can limit the impact of the vulnerabilities described earlier as the software may be constrained from executing code that is not allowed by a security policy. Enforcement of this security policy is achieved by implementing wrappers around relevant system calls and by linking the wrapped versions at load- or run-time. To prevent protected programs from circumventing the policy, limited software-based fault isolation is employed to ensure that the application does not perform direct jumps to unchecked system calls. The policy-enforcement used in Naccio will not stop injected code as only wrapped system calls are checked at run-time, while injected code would probably use the unwrapped ones. Neither does it prevent buffer overflows as generally, system calls are not needed to overflow a buffer. However the ideas presented here could easily be incorporated in such a defense by intercepting the calls at run-time instead of wrapping them or by modifying the actual system calls directly.

### Goldberg, Wagner, Thomas and Brewer (Janus)

Janus [53] is a mechanism that monitors untrusted applications and disallows system calls that the application is not permitted to execute. The Janus framework will read which modules should be loaded from a configuration file. Each of these modules can decide which system call execution it wishes to approve or deny or on which system calls it does not wish to comment. Subsequent modules in the configuration file can reverse earlier decisions made about approving or denying system calls unless the original module specified its decision as final. The decision on whether or not to allow a specific system call is made based on the arguments to the system call: e.g. a call to open could be a harmless request to open a file the application is allowed to access or could be a request to open a sensitive file on the hard disk.

This framework allows an administrator or developer to implement a security policy for an application at deployment-time that can prevent an attacker who is able to inject arbitrary code into the application from executing system calls that the application would not normally need to execute.

The Janus framework starts by reading the configuration file and building a list of most system calls (some system calls like read and write are always allowed for performance reasons) and associated values, called the dispatch table. This dispatch table is then used at run-time: when the application is executed and arrives at a system call, the dispatch table is examined and a decision on whether to allow or deny the system call is made and, in the case of denial, reported back to the process.

### Provos (Systrace)

Some programs in the family of Unix operating systems require more privileges than the user that is executing them. These are called privileged programs and are generally executed with the full privileges of the owner of that program. However the lack of granularity that is available brings about a major security problem: programs that require administrator privilege to execute a specific system call can execute any other calls with the same privileges. This can cause problems if an attacker is able to inject foreign code into the application and forces the program to execute this using one of the vulnerabilities described in section 2. Systrace [95] introduces a way of eliminating the need to give a program full administrator privileges, instead allowing finer grained privileges to be given to an application: which system calls can be executed and with which arguments, i.e. much the same functionality as is offered by Janus.

As defining a policy of which system calls a program is able to execute is complex, Systrace offers a training ability, allowing a user to run the program while Systrace learns which system calls are executed. This allows Systrace to generate a base policy that can later be refined. It is also possible for Systrace to interactively generate system policies, where the user has to make a policy decision whenever an attempt to execute a system call that is not described by the current policy is performed.

### Prevelakis and Spinellis (FMAC)

FMAC [94] is a tool for Unix-like operating systems that implements a file system to sandbox applications. It runs in two modes:

**passive:** where it gathers information on the files that the application opens and adds these files to an access list.

**active:** where only file accesses to files on the list are allowed. A file access to a different file will return a non-existent file error.

If it is impractical to use the active tool, the file list that was generated in passive mode can be used to create a chroot environment. This means that all files that the application accesses are copied into a directory with the same path as the original files (e.g. /etc/passwd becomes /sandbox/etc/passwd) and the root directory of the application is set to this directory. This will prevent the application from reading any files outside of its sandboxed directory. Using FMAC in active mode brings about some problems: care must be taken when running the application in passive mode that it is used as extensively as possible so that all file accesses are performed during this phase. Another problem occurs when an application creates random temporary files: these change every run and make them hard to specify in the access list. As such some directories can be marked as non-checked, allowing the application to read and write files into this directory at its own discretion.

### Kiriansky, Bruening and Amarasinghe (Program shepherding)

Program shepherding [67] is a technique that will monitor the execution of a program and will disallow control-flow transfers that are not considered safe. Program shepherding can be used for example to ensure that programs can

only jump to entry points of functions or libraries, denying an attacker the possibility of bypassing checks that might be performed before a certain action is taken in a function. Another example of a use for program shepherding is to enforce return instructions to only return to the instruction after the call site.

Program shepherding uses a runtime binary interpreter to monitor the program for control-flow transfers. The shepherding techniques are implemented as an addition to RIO, a run-time dynamic optimizer that runs code in an interpreter. To reduce the overhead of this interpretation, code that is executed often is cached in native form. The authors of program shepherding use the same technique to reduce the overhead of their techniques. Checks are performed once and if code is allowed to pass, it is placed in a cache that is not checked anymore at later dates.

Three techniques are used to implement program shepherding:

- Execution of code is dependent on its origin: unmodified code that was loaded from disk could be considered more trusted than code that was dynamically generated or code that was modified since it was loaded from disk. Depending on these origins certain actions could be allowed or denied.

- Control transfer is restricted: this is important to ensure that attackers can not bypass checks by jumping directly to code they wish to see executed, without executing the sanity checks a function might wish to perform before it executes that code. Control transfers can also be denied based on other criteria.

- An uncircumventable sandbox is implemented: thanks to the control transfer restrictions it is possible to implement a sandbox that can not be circumvented by jumping past the checks. This sandbox can then be used to restrict operations that a particular piece of code can perform. The code can be made to adhere to a specific policy for which checks are inserted into code. To protect the run-time environment of RIO, calls to system calls that modify memory protection are prevented from changing the memory privileges of the pages that contain the RIO's code and data.

## 3.5 Anomaly detection

Many of the techniques that are used for sandboxing 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).

### Forrest, Hofmeyr, Somayaji and Longstaff

Forrest et al. [47] define a method for anomaly detection based on short sequences of system calls. A sliding window is used to record system calls during the training period: in that window a sequence of system calls of a specific size is recorded. The window is slided across a trace of the system calls that an application performs. If the program deviates from these sequences of calls in

subsequent runs an anomaly is detected. The smaller the size of the window, the more options an attacker has for forming valid sequences of system calls while still being able to execute useful code. Some sequences that could never exist in the actual program would also not be recorded as an anomaly when using the short sequence of calls technique. Programs that use many different system calls and in many orders might also cause a problem. The allowable sequences might be so large that the attacker is not really restricted anymore.

### Sekar, Bendre, Dhurjati, and Bollineni

Sekar et al. [107] suggest using finite state automata to model system call sequences. They build an automaton during the training period that models the sequence of system calls that are performed during program execution. The advantage of using an automaton is that control-flow structures like loops and branches are captured in the model, allowing the automaton to make more accurate decisions about the validity of system calls. The main problem when building automata for modeling is deciding if two specific system calls are the same or not, which can lead to the building of inefficient automata. To be able to detect if these two calls are the same, the automaton needs to contain more information about the current program state than just the system call, so whenever a system call is added to the automaton the location in the program where the system call was called is also recorded. This allows the program to decide if two calls to the same system call are actually the same call.

Once the automaton has been built, it can be used for anomaly detection during subsequent runs of the program. At runtime execution of system calls is intercepted and the location from where the call was made is looked up. If a transition can be made from the current state to a new state in the automaton with the intercepted system call then the automaton is placed in the new state. If no transition can be made, an anomaly is recorded. The automaton is then resynched with the program using the program location.

When an anomaly is recorded it isn't immediately reported as this might cause too many false positives, instead an anomaly threshold is defined. When reached, action can be taken. The authors implement this using weighted anomalies, some are more likely to have been caused by a malicious user and should therefor weigh more heavily on the 'anomaly index'.

### Wagner and Dean

Wagner and Dean [126] describe four techniques for detecting anomalous behavior. But unlike the other detectors they gather the information that they need about system calls statically instead of dynamically.

- The first technique is the most simple detection of system calls: the set of all system calls that the application executes is recorded. At runtime the detector checks if the intercepted system calls are in the set of used system calls, if they aren't then an anomaly was detected.
- The second technique builds a non-deterministic finite state automaton of the system call sequence performed by the application using the control-flow graph for the program. As with the technique proposed by Sekar et. al [107] the program location from where the system call is recorded too.

Because the automaton is generated statically it could however contain paths that in reality could never be executed. This static analysis will on the other hand be able to detect all program paths, even the ones that a dynamic one might miss if they are not followed during the training stage.

- The third technique attempts to eliminate program paths that would never be executed by augmenting the information with the state of the call stack. The model is extended so the system call sequences can form a context-free language. This extended model is then represented as a non-deterministic pushdown automaton. Because the call state is also tracked, the precision of the model is increased: impossible paths are rejected by the automaton.
- The final technique is the one described by Forrest et al. [47] but using static techniques instead of dynamic ones to build the model.

Wagner and Dean also define an attack against anomaly detectors: mimicry attacks. 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*[4] 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.

## 3.6 Compiler modifications

Compiler modifications encompass a wide variety of solutions and countermeasures: the compiler is a place where many methods to prevent overflows can be added without modifying the language in which vulnerable programs are written. The compiler decides what the C-code will look like when it is compiled and is an important player in generating the execution environment of the program (however, the run-time environment provided by the operating system and run-time libraries are important players too). The constructs that are used by the compiler (e.g. function calls using a stack and return address) are generally the ones that an attacker will try to exploit to gain control over the program's execution-flow. These constructs, together with some assumptions made by the run-time system are what we will call the machine model. As the compiler generates the code that relies on this model, it is also the ideal place to add protection for it.

This section will discuss bounds checking solutions that prevent buffer overflows completely and countermeasures that protect a program's machine model to prevent an attacker from abusing vulnerabilities.

### 3.6.1 Bounds checking solutions and countermeasures

Bounds checking is a foolproof way of eliminating buffer overflows; if size information for every array is available at run-time and is used to perform run-time checks to ensure that code can not write beyond the bounds of an array, then it is impossible for it to overflow.

[4]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.

### Kendall (Bcc)

Bcc [65] is a source-to-source translator used to perform bounds checking on pointer dereferences and array accesses. The source is modified to include calls to functions that perform bounds, null and alignment checking when pointer dereference or array access is performed and to insert wrappers to library functions that are often misused. Alignment checking is done to make sure that a pointer access is properly aligned. Checking for integer overflows is also performed when doing pointer arithmetic. The main disadvantage of using bcc is that the compilation time is increased, more code is generated and the run-time execution of the program is severely slowed down. However, because the low-level pointer representation remains unchanged, code transformed by bcc will still be compatible with existing non-instrumented code.

### Steffen (RTCC)

RTCC [119] is a modification of the Portable C Compiler that adds run-time array subscript and pointer bounds checking. To implement this, pointers are represented by 3 times their normal value, containing the current value of the pointer and the memory addresses of its lower and upper bounds. Since pointers are three times the normal size, they must be converted to a normal pointer when used in system calls. As such all system calls are wrapped so that these pointers can be converted. These wrappers also contain extra checks that make sure that the supplied arguments contain values that the system call expects. The pointer size also poses problems when using integers as pointers; as they are no longer the same size, they must be explicitly cast to a pointer by the programmer. To simplify this task, the compiler comes with a program that will find the places where an integer is used but a pointer is expected and will report the locations to the programmer.

### Austin, Breach and Sohi (Safe C)

Austin et al. [7] implement a bounds checking solution for C. They define a kind of safe pointer that contains the following attributes: value, pointer base, size, storage class (heap, local, global) and capability (forever, never). The value attribute is the actual pointer, the base and size attributes are used for spatial checks (writing outside the array bounds). A valid pointer access lies between the base and the base + size range. The storage class and capability attributes are used for temporal checks (is the memory still allocated). Pointers to global objects will receive the capability forever as they will never be freed, pointers to objects that are freed receive the capability never which will cause an error when they are dereferenced. The malloc and free operations are modified to respectively set and check these attributes.

When an attempt is made to dereference a safe pointer, first a check is done to see if this pointer is still a valid one and afterwards a check is done to make sure that the pointer does not access memory past its bounds.

### Jones and Kelly

The bounds checking solution proposed by Austin et al. [7] is not backwards compatible, meaning that code that was compiled with a different compiler can

not be linked to code compiled with the Austin bounds checking compiler. Jones and Kelly [62] propose a bounds checking solution which is backwards compatible: the pointer representation is left unchanged so code compiled with this bounds checking compiler is compatible with code which was not. To accomplish bounds checking the compiler keeps a table of all known storage objects that maps pointers onto a descriptor of an object that contains the base and size of that object. Whenever a pointer is dereferenced or arithmetic is done on a pointer, a check is done to ensure that the pointer does not point outside the object to which it is referring. Whenever an object is created inside checked code it is added into the object table. Because the pointer representation is kept unchanged, pointers can be be passed in and out of non-checked and checked code without requiring modifications. Bounds checking is done when pointer arithmetic is performed, however the ISO C standard [2] allows generation of a pointer that is one past the end of an array as this often occurs at the end of a loop. To prevent the compiler from flagging this as an error, every array is padded with an extra byte.

#### Suffield

Suffield [121] extended the Jones & Kelly compiler to support both C and C++ and makes some different implementation choices. Jones & Kelly implement the checking in the C language front end which makes it harder to implement the checker for both languages without significant code duplication. As such Suffield moved the instrumentation to the level of the compiler where the intermediate representation of the program is generated.

The author also decided to allow the generation of pointers which point further than one element past the bounds of the object, so a different approach to supporting out of bounds pointer generation was taken. When an out of bounds address is generated, one byte is allocated on the heap and is stored in the object tree with information about the object it is derived from, the offset from the end of the object and the fact that it is an out of bounds object. The address of this byte is then returned as the result of the arithmetic operation. When this pointer is subsequently used for other arithmetic operations, it is first looked up in the object tree. If it is present, the real address of the object plus the offset stored in the tree is used. The same technique is used when doing pointer comparisons.

#### Lhee and Chapin

Lhee and Chapin [78] propose using compiler and library modifications to do bounds checking. Their approach consists of a modification to the preprocessor to process the source file with debugging information turned on and then using the extra debugging information to generate a type table that associates each function address with information about its local buffers, i.e. their sizes and offsets from the stackframe. The type table is also extended with the addresses of static buffers and their sizes. And finally, at run-time this type table is updated with the same information for dynamically allocated objects.

A shared library is loaded at run-time that intercepts calls to vulnerable string and memory copying functions and does bounds checking on the arguments using this type table. To find the size of a stack-allocated buffer at

run-time the following algorithm is used: the return address of the previous stackframe is located (where the vulnerable copying function was called) and is then used to find the size of the buffer by looking up the function address in the type table.

### Oiwa, Sekiguchi, Sumii and Yonezawa (Fail-Safe C)

The aim of the Fail-Safe ANSI-C compiler [86] is to be a compiler that supports the full ANSI C standard but prevents behavior that is marked as undefined by the standard. It is thus a type and memory safe compiler for C. Every contiguous memory region also contains its size and type information. In this compiler a pointer is represented by a collection of three values: the base address of the contiguous region it is pointing to, the offset to that region and a cast flag that indicates if the pointer may refer to a value other than its static type.

To prevent the dereferencing of an invalid pointer, bounds checking, using the size of the region that the base of the pointer points to and the pointer's offset, and type checking is done before the pointer is dereferenced. The integer interpretation of a pointer is it's base + offset. When a pointer is cast to an integer the three value representation is copied to the integer, allowing the integer to be cast back to a pointer at a later date, meaning that both pointers and integers are of the same size (for normal integers the base value is 0).

Dangling pointer references are avoided by marking freed memory regions as released but not actually freeing them, allowing dangling pointer references to be caught. The memory is eventually deallocated by a garbage collector.

### Ruwase and Lam (CRED)

CRED [101] is a bounds checking countermeasure that is also based on the solution provided by Jones and Kelly [62]. However it relaxes the ANSI C standard in that it allows the generation and dereferencing of out of bounds addresses[5], because many programs use out of bounds addresses in calculations and comparisons but do not cause buffer overflows and would fail if too strict checking is applied.

The approach taken by CRED is to create a unique out of bounds object for ever out of bounds address value that is stored and to substitute the value of that address with the address of its OOB object. This object contains the out of bounds address and the object that it refers to. As with the Jones&Kelly compiler an object table is kept and if a pointer is dereferenced and can not be found in the table and isn't an unchecked object (non-instrumented code) an error will be raised. The OOB objects are kept in an OOB table and when pointers are used in arithmetic or comparison and can't be found to point to an object in the object table or to an unchecked object, the value is looked up in the OOB table and subsequently the OOB-stored value is returned for use in the operation.

To improve performance CRED does not bounds check every array but only the ones containing string data as these are the ones that are usually involved in buffer overflows.

[5]The ANSI C standard allows generation of the address immediately past the end of the array because this is often used to determine the end of an array in a loop, however dereferencing this location is still undefined as is generating other out of bounds addresses.

### 3.6.2 Protection of stackframes

These protection mechanisms attempt to protect information stored on the stack from being exploited by a buffer overflow. This type of protection comes from the observation that the most commonly executed buffer overflow attack is an attack on a stack-based buffer where the buffer is overflown to overwrite the return address of a function. As such most countermeasures focus on protecting the return address, but have extended the protection to also protect other stack-stored information that could be used by an attacker to gain control over the execution-flow of a program (e.g. the frame pointer or regular pointers stored in local variables).

#### Cowan et al. (StackGuard)

StackGuard attempts to protect the return address that is stored on the stack by placing a canary before the return address [27, 29]. In the function prologue, either a random canary (a 32 bit random value that is generated at program startup) or a terminator canary (a 32 bit value containing NULL, LF, -1 and CR: 0x000aff0d). One of the values in the terminator canary is generally used to mark the end of input for a specific vulnerable string function. When the function exits, the function epilogue will check if the canary has been changed. If it has changed, an alert is logged and the program is terminated, if it hasn't the canary is removed from the stack and the function is allowed to return as it would normally. Through indirect pointer overwriting (see section 2.1.2) this countermeasure can be bypassed by using the pointer to overwrite the return address directly without modifying the canary. To remedy this, a modification to StackGuard was proposed [25] where the return address would be protected by using random values. Instead of using those values as canaries they are xor-ed with the return address to form the canary (named xor canaries). This will protect the return address, but will not prevent the attacker from overwriting other interesting memory locations. Ricarte [96] describes additional techniques for bypassing StackGuard. One technique overwrites the arguments to a function: if a pointer is passed as argument, indirect pointer overwriting can be used to overwrite any memory location. This can be abused by an attacker to overwrite the memory location of library calls [6] that the function (or StackGuard's checking functions) may call after overwriting the pointer. Ricarte also describes other techniques to bypass StackGuard by overwriting frame pointer (see section 2.1.2).

#### "Vendicator" (Stack Shield)

Stack Shield also attempts to protect the return address that is stored on the stack from modification [123].

It has two methods available for doing this [140]:

[6] Dynamically loaded libraries are implemented as Position Independent Code (PIC) that does not rely on being loaded at a specific location. To execute calls to library functions the program uses a Global Offset Table (GOT) to locate the functions it requires. Overwriting an entry in the GOT will cause the program to look for the function at the new location. Actually the whole process of function lookup for dynamically loaded code is more complicated but out of our scope, see [77].

**Global ret stack method** At program startup, a global array, called the return value table is allocated on the heap. In the function prologue, the function's return address will be copied into this return value table. In the function epilogue the return address on the stack will be replaced (or compared with, depending on the option specified) with the one stored in the return value table, a return value table pointer will be decremented and the function will return as usual. The return value table is static in size, if more function calls get done than the size of this array (i.e. the return value table pointer is larger than the return value table top), these new function calls will not be protected, but the return value table pointer will be incremented. When the return value table pointer gets smaller than the return value table top again, subsequent function calls will be protected again.

**Ret range check method** Another method of protection that Stack Shield offers is to place a global variable at the beginning of the data section. In the function epilogue the value of the return address is compared to the address of this variable: if the return address is lower, it means that it points into the text section and it is safe to return. If its higher, it means that the return address is pointing into the data segment and the program is terminated. Besides protecting the return address there is also a possibility to use this method to protect function calls (to combat function pointer overwriting).

The protections offered by Stack Shield can also be bypassed using the techniques described by Bulba and Kil3r [18] and Ricarte [96].

### Chiueh and Hsu (RAD)

Return Address Defender (RAD) uses the same approach as Stack Shield, it copies the return address into a Return Address Repository (RAR) in the function prologue [21]. Before returning from the function, in the function epilogue, the stored address is compared to the actual address, if they are the same then the function is allowed to return to this address. Besides this protection, it attempts to protect the RAR from modification too. To accomplish this it supports two kinds of protection:

Minezone RAD: A global integer array is declared as RAR, but the pages at the beginning and end of the RAR are marked as read-only (minezones). This prevents overflows of other heap-based arrays from overwriting the middle of the RAR, where the return addresses are actually stored.

Read-Only RAD: The whole RAR is marked as read-only and is only unmarked read-only in the function prologue when a new address is placed into the RAR.

The system calls setjmp and longjmp might cause problems with RAR-stack inconsistency: longjmp performs a jump to the address where the last setjmp was executed. When the longjmp is executed, several stackframes might be popped to return to the setjmp address. This could cause an inconsistency between the RAR and the stack: some return addresses have been removed from the stack that are still at the top of the RAR, causing a mismatch and a

false exploitation attempt to be reported. When such a mismatch occurs, RAD will keep popping values off the RAR until either a match occurs or the RAR is empty. When the RAR is empty and no match has occurred, then it is safe to assume that the return address was modified. However since the return address can be any valid return address that is currently on the stack, this opens RAD up to a possible attack, where the attackers, instead of injecting their own code, can make the program execute code at any of the return addresses currently on the stack.

### Etoh and Yoda (Propolice)

Propolice [39] is based on the same ideas as StackGuard, but it extends the provided countermeasures to try and prevent the attacks described in [18, 68, 96]. The first difference with StackGuard is that it places the canary before the frame pointer instead of before the return address to prevent an attack where only the frame pointer is modified [68] and the return address is left unchanged. To prevent the other attacks a few changes are made, firstly only a random XOR canary is supported to prevent the Emsi attack [25]. To prevent the attacks described in [18, 96] the organization of the stackframe is changed: all arrays are placed before any other local variables on the stack, meaning that arrays are the closest to the canary. This will prevent an array from overflowing a pointer that was declared before it in the function and using it to overwrite values stored at specific addresses. A function is also protected from argument overwriting by making a local copy of pointer arguments.

### Bray et al. (Microsoft .NET Stack Protection)

Microsoft has added a security mechanism to Visual Studio .NET (often referred to as the /GS compiler flag [54]). This mechanism works in much the same way as StackGuard: a random security cookie is placed on the stack next to the saved frame pointer to protect both the saved frame pointer and return address of a function [17]. This will protect the application from normal buffer overflow attacks, it will however not protect it from the attacks described in [18]. While the attacks in [18] require a specific situation in the code (i.e. a function pointer must be located on the stack before the overflown array and subsequent user-controlled changing of the pointer), a generic way to defeat the protection exists and is described in [80].

Windows has a built-in exception handling system for every application. Even if the application does not register any exception handlers, there will be a default handler that is set up on thread initialization. The information about the exception handlers is stored on the stack in a linked list. As such, attackers can overflow a buffer, the security cookie, return address and can keep on overflowing until they have overwritten the exception handler's pointer. The attackers must then cause an exception before the function returns, as they overwrote the security cookie. This can easily be done by attempting to write past the end of the stack. However some countermeasures have been implemented: exception handlers in a module are registered and their addresses are stored, before calling an exception, the handler's address is compared to the list of stored addresses and if no match is found, the exception will not be executed. If, however, the handler's address points outside the module's address space then it will still

be executed (however it may not be on the stack). This allows an attacker to bypass the protection by pointing the handler to a piece of code, outside of the address space of the current module, that will jump to the attacker's code.

### 3.6.3 Protection of all pointers

All of the current attacks that make a program execute injected code occur by corrupting code pointers. An attacker must modify a code pointer to point to their injected code so that the program will at some stage, when calling this code pointer (expecting it to point to legitimate code), execute the attacker's code.

#### Cowan, Beattie, Johansen and Wagle (PointGuard)

PointGuard [28] attempts to protect pointers by encrypting them while they are in memory and decrypting them when they are loaded into registers where they are safe from overwriting. While an attacker will still be able to modify pointers stored in memory, when the pointers that the attacker provides are decrypted, they will point to different (possibly inaccessible) memory locations. Without the encryption key, the attacker is unable to use pointers that will decrypt to a predictable value. The encryption is extremely simple so as not to cause a significant overhead to execution time of the program. A random value is generated at program start-up and every pointer is XOR'ed with this value, this will generally make it impossible for attackers to bypass the protection as the process will most likely crash when a pointer is decrypted wrongly and subsequently used. When the program is restarted a new key will be generated, making it very hard for attackers to guess the key. However, if attackers are able to cause the program to print out pointer information through some other vulnerability then they might be able to guess the decryption key and would then be able to encrypt their own pointers correctly.

#### Yong and Horwitz (Security Enforcement Tool)

The Security Enforcement Tool suggested by Yong and Horwitz [138] is a tool that adds run-time protection against invalid pointer dereferences on pointers that will be dereferenced for a write operation. It is implemented as a source-to-source translation that adds run-time checks to pointers after determining, through static analysis, that they are possibly unsafe. The run-time checks are added by keeping a status bit for every byte in memory (called a mirror). The status bit determines if writing to a specific memory region via an unsafe pointer is allowed (appropriate) or not (inappropriate).

The static analysis performed by the tool is divided into three steps: firstly a flow-insensitive analysis is performed to gather points-to information (the set of variables that a variable points to at some time during execution). This points-to analysis is used to determine which variables are possibly unsafe: pointers that may point to invalid memory and whose memory location is written to or freed. In the final step, tracked variables are identified, these are the correct variables that at some time during the execution may have an unsafe pointer referring to them. Subsequently the program is instrumented, at every place of allocation of a tracked variable, the mirror is updated to mark the memory as

appropriate in the mirror, when deallocation occurs, the memory is marked as inappropriate. Finally, before each dereference for writing or free operation, the mirror for the target memory is checked for appropriateness. If a write or free is done to memory that is marked inappropriate, the program will be terminated.

## 3.7 Operating system and hardware modifications

The observation that most attackers wish to execute their own code has led to many proposed countermeasures that will not try to prevent a vulnerability or its exploitation but will try to prevent execution of injected code. The text will focus on the IA32 architecture mainly because most attacks and solutions have been implemented for this architecture.

### 3.7.1 Non-executable memory

Most operating systems divide process memory into at least a code (also called the text) and data segment and 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). As such attackers have to inject their code into the data segment of the application. As most applications do not require executable data segments as all their code will be in the code segment, some countermeasures suggest marking this memory as non-executable, which will make it harder for an attacker to inject code into a running application.

#### "Solar Designer" (Non-executable stack)

Marking the stack as non-executable is a way of preventing buffer overflows that inject their code into a stack-based variable (usually the buffer that is being overflowed): the processor or operating system will not allow instructions to be executed in this specific memory region. In the implementation of a non-executable stack by Solar Designer for the Linux kernel on the IA32 architecture [116] (a detailed analysis can be found in [140]), the code segment size limit is reduced from the default 4GB to 3GB-8MB. Since the stack in Linux on the IA32 starts at 3GB, is 8MB large by default and grows downwards, the stack will be marked as non-executable. Marking the stack non-executable can cause programs that rely on its executability to break (gcc compiled programs that use nested functions, some lisp compilers, the Linux kernel signal handling, ...). While setting the stack non-executable is a zero-cost operation (it is done at kernel load-time), it is also easily bypassed [134]. Instead of injecting code on the stack and then pointing the return address to this code, the desired parameters are placed on the stack and the return address is pointed to existing code (a simple example is to call the libc wrapper for the system system call and to pass it an executable that will execute the attacker's code as argument).

#### "The PaX Team" (PaX PAGEEXEC)

PaX [122] is a kernel patch for Linux that implements a non-executable stack and heap through non-executable pages for the IA32 architecture (which has no native support for it). Hardware support for non-executable pages would stop

an attacker from injecting foreign code into the application as on most platforms the text section (that contains the code for the application) is marked read-only.

Because the IA32 architecture does not natively support non-executable pages, the patch uses one of the page privilege flags to implement it. It marks all pages which should be non-executable as requiring superuser privileges: whenever an application tries to access one of these pages a page fault exception will be generated. PaX will intercept the exception and check if the processor attempted to execute an instruction in this page or if an access of data was attempted. In the case of an instruction execution attempt, the application will be terminated. If data access was attempted, the page will be given user privileges, a dummy access to the page will be performed so that it is stored in the data cache and then the page will be marked as requiring superuser privileges again. A more detailed analysis is available in [140].

### 3.7.2 Randomized instruction sets

Another technique that can be used to prevent the injection of attacker-specified code is the use of randomized instruction sets. 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've 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. Both implementations that we will describe here incur a significant run-time performance penalty when unscrambling instructions because they are implemented in emulators, but it is entirely possible, and in most cases desirable, to implement them at the hardware level and thus reducing the impact on run-time performance.

#### Barrantes et al. (RISE)

Barrantes et al. [9] implement a proof-of-concept implementation of these randomized instructions by using emulators that emulate a processor. RISE is built on Valgrind an x86-to-x86 binary translator that is used to detect memory leaks. RISE scrambles the instructions at load-time and unscrambles them before execution. By scrambling at load-time the binaries can remain on the disk unmodified and as such the scrambled binary can not be examined by a potential attacker.

#### Kc, Keromytis and Prevelakis

The approach by Kc et al. [64] is also a proof-of-concept implementation using an emulator. It mainly differs from RISE in the fact that binaries are scrambled on disk instead of at load-time. The key is then extracted from the header at load time and stored in the process control block (PCB) [7]. The key is then stored in a special register that can only be written to with a special instruction (and can't

[7] A data structure stored in kernel memory containing information on the process that the process itself can't access or modify.

be read from). Storing the encrypted binaries on disk has some major drawbacks though: an attacker that has local access could examine the encrypted binary and extract the key. More importantly the current implementation can only work with statically linked binaries because dynamically loaded libraries would have to be encrypted with the same key as the executable which means that any programs using the libraries would also have to use the same key. As a result the entire system would probably be using just a single key for all programs.

### 3.7.3 Randomized addresses

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. 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.

#### "The PaX Team" (PaX ASLR)

The PaX kernel patch [122] that offers a non-executable stack and heap, which were described in a previous section, also offers address space layout randomization (ASLR). It expects the compiler to generate position independent code and randomizes the specific bases addresses of specific memory segments such as the stack, the heap, the code and the memory mapped segments of a program when it loads the program into memory. However if attackers are able to cause the victim program to print out part of its memory, they might be able to deduce the base address of the desired segment and could modify their attack accordingly.

#### TRR

While not a kernel modification, TRR [136] can be seen as an operating system modification. It modifies the dynamic program loader in Linux, which is used to load programs in to memory for execution. It randomizes many of the memory locations where an attacker would place his code or that an attacker would modify to execute code: the user stack set up by the kernel is randomized by allocating a new stack at a random location below the current one. The contents is copied to the new one and the pointers to it are adjusted accordingly. Finally the stack pointer is set to point to the new stack and the old stack is freed. The heap is randomized by randomly growing it. The location of the shared libraries are randomized by allocating a random-sized region before loading the shared libraries into the program's address space.

The Global Offset Table (GOT), which is used to locate the information needed to execute position independent code (PIC) and is used by the PIC to access data, is loaded into a fixed position. To randomize the GOT, all positions that access the GOT directly must be rewritten to access this new position.

#### Chew and Song

Chew and Song [20] propose another operating system randomization: the randomization of system call mappings, making it harder for a remote attacker to

call system calls in injected code. Most dynamically applications will call system calls through libc wrappers, for other code the system call numbers could be rewritten on the fly at load-time or generated correctly at compile-time. The same technique is also suggested for libc calls, their location is changed and the original program is rewritten by the dynamic program loader. The stack location is modified by allocating a random size region on it at program load time.

Another change to the operating system is also suggested: timed capabilities. Instead of verifying permissions for a system call to open a file once when the program starts, the program would have the ability to add a timeout to the call to open. If the timeout has passed, then the permissions are rechecked, if they are no longer available an error will occur.

### 3.7.4 Protection of return addresses

As most buffer overflows attempt to overwrite the return address of a function, many countermeasures focus on protecting this return address, but compiler modifications where run-time checks are added to verify the correctness of the return address can be detrimental to system performance. This section will examine a hardware modification that protects return addresses from changes or detects changed return addresses.

#### Xu, Kalbarczyk and Iyer

Xu et al. [137] suggest two different countermeasures to defend against buffer overflow attacks:

**Splitting the control and data stack** is a straightforward approach where the return addresses are stored on one stack and the function data (i.e. arguments, local variables, etc.) are stored on the other stack. This splitting can be done on both the software and hardware level: the authors have implemented this as a compiler modification where extra code is generated in the function prologue to save the return address to a control stack and in the function epilogue where the return address is copied from the control stack back to the normal (data) stack. For the hardware-supported version the changes needed to make the IA32 architecture compatible with this approach are also described: an extra register called csp (like esp, but for the control stack) is needed and the call instructions need modification (the return address would be pushed onto the control stack), the ret instruction would then pop the return address from the control stack.

Multithreadedness is handled in the same way as is done currently for multithreaded applications: each thread has its own stack and in this case has both a separate control and data stack.

**The secure return address stack (SRAS)** on the other hand attempts to detect buffer overflows occurring in processors with support for a return address stack. These processors predict the return address when a call instruction is fetched by the processor to allow the processor to predict which instructions to prefetch when executing a return [110, 129]. Three possible extensions to the RAS mechanism are proposed to detect buffer overflows:

1. If a RAS misprediction occurs, i.e. the actual return address is different from the return address that was predicted by the processor, it will cause an exception when such a misprediction occurs, allowing the exception handler to verify if a buffer overflow really did occur. This will however cause a significant performance hit if the processor mispredicts (the prediction mechanism is not 100% accurate).
2. To prevent this, keeping a non-speculative RAS is proposed: the updating of the RAS is done at the commit stage as opposed to the fetch stage of the processor. This will cause mismatches between the RAS and the return address that is stored on the stack only when the RAS overflows.
3. To prevent overflowing and causing a mismatch of the return addresses on the RAS and the actual return addresses, the RAS is copied to memory on overflow. When the new RAS subsequently underflows the memory-stored RAS is restored.

#### Özdoğanoğlu et al.

Similar SRAS suggestions have been made by Özdoğanoğlu et al. (SmashGuard) [90] and Lee et al. [76]. Mostly differing in how the system calls setjmp and longjmp are handled. The SmashGuard authors suggest a similar technique as the one used by RAD: when a mismatch occurs, SmashGuard will move down the stack until it finds the correct value or reaches the end of the stack. However this will stop at the first match for the return address while the actual return address after that particular longjmp could be further down the stack. To combat this, the authors suggest storing the frame pointer too and matching both.

#### Lee, Karig, McGregor and Shi

Lee et al. offer four possible options for handling setjmp and longjmp calls. The first option is to prohibit the calls. The second option is to allow extra instructions that allow the compiler to manually push and pop values from the SRAS when generating code for longjmp and setjmp (or similar) calls. The third option also makes use of the SRAS push and pop instructions, but suggests injecting the calls at run-time by using a software filter that dynamically identifies calls to setjmp and longjmp. The final suggested option is to allow users to turn off the SRAS for a particular application.

#### Frantzen and Shuey (StackGhost)

StackGhost [49] is an operating system modification designed for the SPARC architecture. On this architecture function calls are not always handled via the stack, but in most cases (if nesting or recursiveness is not deep), register windows are used. The SPARC [89] has 8 global integer registers that are visible from anywhere within a program. A function will have 24 available integer registers (called a register window) that are divided into 3 groups: input registers, local registers and output registers. To improve execution time of function calls, the processor supports register windows: when a function call is executed, a new window is allocated with the output registers of the calling function becoming

the input registers of the called function and 16 new registers being allocated for the called function. This speeds up execution as the processor must not write register data to the stack and retrieve it on function return. When the processors runs out of registers (it usually has enough for seven or eight windows), it will generate a register window overflow interrupt and the operating system will write the oldest window to the stack. As long as register windows are available, it is not possible for an overflow to overwrite the function's return address or frame pointer as they will still be contained in registers. However when the oldest window is saved to the stack, they are again vulnerable to overwriting. The authors suggest solving this by encrypting the return address, but this could either be trivially broken or could be detrimental to system performance. They therefor also suggest a solution that makes use of a hashtable stored in the Process Control Block (a kernel structure). Whenever a register window is saved to the stack, the return address is saved into the hashtable and a random value (that is used as key for the hashtable) is written to the stack instead of the address. When the window is restored, the return address is looked up in the hashtable using the stack-stored random value.

## 3.8 Library modifications

### 3.8.1 Protection of stackframes

#### Baratloo, Singh and Tsai (Libsafe and Libverify)

Libsafe [8] replaces the string manipulation functions that are prone to misuse with functions that prevent a buffer from being overflown outside its stackframe. This is done by calculating the size of the input string and then making sure that the size of the source string is less than the upper bound of the destination string (the space from the variable's stack location to the saved frame pointer). If it is not smaller, the program will be terminated. Again, as is the case with several other countermeasures, this protection can be bypassed using indirect pointer overwriting.

Libverify [8] offers the same kind of protection as Stack Shield: upon entering a function it saves the return address on a return address stack (that it calls a canary stack) and when exiting from a function the saved return address is compared to the actual return address. The main difference with Stack Shield lies in the way that this check is added: Libverify does not require access to the source code of the application, the checks are added by dynamically linking the process with the library at run-time.

Libverify will copy the function to heap memory and will overwrite the first instruction of the original function with a jump to Libverify's function entry code that saves the return address to the canary stack. The rest of the original function is overwritten with trap instructions to prevent absolute jumps from transferring control to the original function. The trap handler will return the execution-flow to the copied function. Finally it will also overwrite the return instruction of the copied function with a jump to Libverify's function exit code that will check if the return address is unchanged. If it has changed, the program is terminated.

#### Snarskii

Snarskii [114] wrote a patch to the FreeBSD implementation of libc that does an integrity check before returning from several libc string copying functions (strcpy, sprintf, gets) which are known to be vulnerable to buffer overflows if misused. Before returning from these functions, the patch checks to make sure that no stackframes are contained in the memory range that was written to by one of these functions.

#### Snarskii (Libparanoia)

Libparanoia [113] is an extension of the idea of the Snarskii integrity patch by the same author. It also modifies the vulnerable libc string copying functions: the frame pointer and return addresses are saved to the data segment (with a maximum of ten frames deep being saved), then the string manipulation is performed and afterwards the frame pointers and return addresses are compared to the stored values, if any have changed the program is terminated.

### 3.8.2 Protection of dynamically allocated memory

#### Robertson, Kruegel, Mutz and Valeur

This countermeasure [99] attempts to protect against attacks on the dlmalloc library management information. This is done by changing the layout of both allocated and unallocated memory chunks. To protect the management information a checksum and padding (as chunks must be of double word length) is added to every chunk. The checksum is a checksum of the management information seeded with a global read-only random value, to prevent attackers from generating their own checksum. When a chunk is allocated the checksum is added and when it is freed the checksum is verified. Thus if an attacker overwrites this management information with a buffer overflow a subsequent free of this chunk will abort the program because the checksum is invalid.

#### Krennmair (ContraPolice)

ContraPolice [70] also attempts to protect memory allocated on the heap from buffer overflows that would overwrite memory management information associated with a chunk of allocated memory. It uses the same technique as proposed by StackGuard, i.e. canaries to protect these memory regions. It places a randomly generated canary both before and after the memory region that it protects. Before exiting from a string or memory copying function a check is done to ensure that if the destination region was on the heap that the canary stored before the region matches the canary stored after the region. If it does not the program is aborted.

#### Perens (Electric Fence)

Electric Fence [91] is a debugging library that replaces the default implementation of the malloc library with its own version that changes the behavior: every time a portion of memory is allocated it places an inaccessible virtual memory page either immediately preceding the allocated memory or immediately following it. This results in an immediate error if an under- or overrun occurs.

When memory is freed, the page it is contained in will be marked inaccessible to detect dangling pointer references. This approach is not meant to be used in a production environment as any memory allocation will take up two virtual memory pages (for a total of 8KB on IA32 machines) and as this memory will never be freed, the resulting program uses an enormous amount of resources.

#### Fetzer and Xiao (HEALERS)

HEALERS [42, 44, 43, 45] is a fault-containment wrapper that is designed to improve the security and robustness of an application. It takes a number of measures to do this: calls to the C library are intercepted and replaced with versions that do checking on arguments to ensure correct use of the API. It can also wrap calls to other libraries, by first using fault injection to gather information on the correct use of the library and to generate wrappers that will perform checks to ensure that the application can not call the API with arguments that would cause it to misbehave.

In [42] the wrapper is used to protect programs from attacks using heap-based overflows. It replaces all C library functions that could be used to overflow a heap-based buffer with a bounds checked version. Wrappers are also implemented for the malloc and free functions: when memory is allocated, the size and position of the memory are recorded into a table and when the memory is freed, the information is removed from the table.

Overflowing of heap-allocated memory using C copying functions is detected by first checking if the area written to is on the heap and then by comparing the range of the memory region with the area that will be written to. If the area lies outside the range, an overflow would occur and the program is terminated.

### 3.8.3 Format string countermeasures

These countermeasures attempt to prevent format string vulnerabilities by intercepting calls to format functions and by performing sanity checks before executing the format function.

#### Cowan et al. (FormatGuard)

FormatGuard [26] offers countermeasures for the format string attacks described in section 2.4. These countermeasures are based on the observation that most format string attacks have more specifiers in the format string than arguments passed to the format function. FormatGuard tries to prevent format string attacks by counting the amount of arguments that a format string expects and compares this to the amount of arguments that were actually passed to the function. If more were expected than provided it is likely that someone is attempting to exploit to a format string vulnerability and the program will be terminated.

#### Robbins (Libformat)

Libformat [98] is a library that intercepts calls to format functions and checks the supplied format string. If the string is located in a writable segment of the program and contains the %n specifier then the program is terminated.

### 3.8.4 Safer libraries

Most buffer overflows occur due to misuse of the standard C string manipulation functions. Safer libraries attempt to prevent such vulnerabilities by offering developers new libraries for string manipulation that are less prone to misuse.

#### Miller and de Raadt

To solve the problems associated with the standard string manipulation functions (strcpy, strcat, sprintf, ...), bounded string copying functions exist. These functions accept an argument which denotes the maximum size that may be copied to the destination string. However problems can occur with these functions too, for example a common mistake is to assume that all destination strings passed to strncpy are NULL-terminated, this is only true if the size of the source string is smaller than the size parameter. Strncpy also behaves differently than the standard strcpy implementation: if the size of the source string is smaller than the size parameter, the rest of the destination string is filled with NULL bytes causing a loss of performance that the standard strcpy does not suffer from. Strncat has a similar problem: it guarantees NULL termination but the size parameter that is passed to it must take this into account. Strlcpy and strlcat [84] attempt to solve this inconsistency by offering C programmers string manipulation functions, that accept a size parameter, are guaranteed to be NULL-terminated and do not fill the remaining memory with NULL bytes.

#### Messier and Vega (SafeStr)

The SafeStr[81] library is a replacement string library for C that is immune to buffer overflows and format string vulnerabilities. Strings are no longer defined by a an array of characters but are defined as being of type safestr_t (which can be cast to a char * and back). This safestr type has extra accounting information on the actual length and the allocated length of the string.

For most of the C string manipulation functions the safestr library has a safe counterpart that will do bounds checking using the safestr type accounting information. Format string vulnerabilities are stopped by having safe versions of printf that do not allow the use of the %n specifier and does sanity checking before printing the string. The safestr library also supports a kind of tainting: strings can be flagged trusted or untrusted and this flag can subsequently be checked whenever a trusted string is needed.

### 3.8.5 Randomized addresses

Randomization of addresses is, next to the kernel-level (see section 3.7.3), also possible at the library-level. The following countermeasure describes such an address obfuscation technique that is implemented at the library-level.

#### Bhatkar, DuVarney and Sekar

Bhatkar et al. [11] describe address obfuscation techniques that make it harder for an attacker to exploit one of the vulnerabilities described earlier. But as opposed to PaX the implementation is not done at the kernel level, but at the library level. They too support the same kind of randomization that PaX

ASLR does, but they extend this idea. Firstly the order of static variables, local variables in a stackframe and the shared library functions is randomized. This makes it harder for an attacker to overwrite adjacent variables. However, some objects can not be moved as the system expects them to be in a specific order. To make it harder for an attacker to overwrite these, random-sized gaps are introduced between objects. For implementation purposes[8] the authors have chosen to implement the randomization at link-time; this has an important shortcoming: the randomization is static, i.e. once the program has been randomized it will, for every execution, always contain the same randomization. This makes it easier for an attacker to figure out the base address, relative distances between variables and order, either by examining the program binary or by exploiting a vulnerability that causes the victim program to print out parts of its memory. For this reason the authors suggest (and plan) to use a different method of implementation that does randomization at the beginning of execution (i.e. different randomizations each time the program is run) and that supports rerandomization at run-time.

[8]Note that the authors mention that not all their suggested techniques have a proof-of-concept implementation

# 4 Synthesis of countermeasures

## 4.1 Categories

We have divided the countermeasures of section 3 into several categories based on the vulnerabilities they address, their limitations in applicability, their limitations in protection, the type of protection they offer (preventive, defensive, mitigating) and finally the type of response they have when a problem is detected. We have summarized the results in tables 6, 7, 8 and 9 for the countermeasures that are used at deployment time. The results for countermeasures that are used at development time are summarized in tables 10 and 11.

### 4.1.1 Vulnerability categories

Table 1 contains a list of the vulnerabilities that the countermeasures in this document deal with. They reflect the scope of the vulnerabilities that the author wished to address. However, in some cases (this is especially true for integer overflows) a countermeasure implicitly offers protection for a certain kind of vulnerability without explicitly wanting to protect against it. In such cases we have also listed these vulnerabilities as being in the scope of the countermeasure. An example of this is bounds checking solutions that offer protection against integer overflows because integer overflows are usually misused to cause buffer overflows. Thus we have listed integer overflows as being part of the protection offered by bounds checking solutions, even though they do not offer explicit protection.

| Code | Category |
|---|---|
| $V_1$ | Stack-based buffer overflow |
| $V_2$ | Heap-based buffer overflow |
| $V_3$ | Dangling pointer references |
| $V_4$ | Format string vulnerabilities |
| $V_5$ | Integer errors |

Table 1: Vulnerability categories

### 4.1.2 Applicability limitations categories

Table 2 lists the codes for specific requirements or limitations in application that some countermeasures may have. For example: some countermeasures require the source code of a program to implement their protection. This is a limitation as to where the countermeasure can be applied: it can not be used to protect programs for which no source code is available.

### 4.1.3 Protection limitations categories

Tables 3 contains the list of codes of known limitations in protection that a countermeasure may have. The countermeasures may have more limitations than the ones in the table, however the limitations here give a good indication

| Code | Applicability limitation categories |
|---|---|
| $A_1$ | Source code required |
| $A_2$ | Explicit manual changes or intervention required |
| $A_3$ | Not compatible with non-instrumented code |
| $A_4$ | Does not compile/analyze all code |
| $A_5$ | Needs hardware modifications |
| $A_6$ | Operating system modifications required |
| $A_7$ | Architecture specific |
| $A_8$ | Linux specific |
| $A_9$ | Dynamically linked executable required |
| $A_{10}$ | Only protects libc string manipulation functions |
| $A_{11}$ | Breaks applications requiring executable memory |
| $A_{12}$ | Only for systems that use the Executable and Linking Format (ELF) |
| $A_{13}$ | Unix systems specific |
| $A_{14}$ | Inadequate support for threading |
| $A_{15}$ | Program needs to be implemented in a special language |
| $A_{16}$ | Requires annotations |
| $A_{17}$ | Only operates on integers |
| $A_{18}$ | Does not scale to larger applications |
| $A_{19}$ | Does not examine string operations |

Table 2: Applicability limitation categories

of what options an attacker may have when attacking an application that was protected with a specific countermeasure.

#### 4.1.4 Type categories

The type of protection that a countermeasure provides are contained in table 4. Countermeasures that offer detection detect an attack when it occurs and take action to defend the application against it, but do 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. Finally the last type of countermeasures are the ones that do not try to detect or prevent an attack or a vulnerability but try to mitigate the damage that an attacker can do when trying to exploit a vulnerability. Countermeasures that make it harder for an attacker to exploit a vulnerability (like randomization techniques) are also of the type mitigation.

#### 4.1.5 Response categories

This section contains the table of categories (table 5) that describe what the countermeasure does when a problem is detected. A countermeasure might be able to restore the program's original state and continue execution or might decide that continuing operation is unsafe and will log a message describing the problem and terminate the application. If the countermeasure's response is to report a problem this means it might just log a problem or it could, in some cases, take an action based on the problem it has reported.

| Code | Protection limitation categories |
|---|---|
| $P_1$ | Does not check for dangling pointer references to the stack, only to the heap |
| $P_2$ | Does not check arrays in structs or other arrays |
| $P_3$ | Passing out of bounds pointers to non-instrumented code may cause unexpected behavior |
| $P_4$ | In some cases the size of buffers can not be determined: stack allocated buffers allocated with alloca and arrays of variable length |
| $P_5$ | Only protects libc functions (e.g. strcpy, ...) |
| $P_6$ | Only strings are bounds checked |
| $P_7$ | Can be bypassed using indirect pointer overwriting |
| $P_8$ | If an attacker is able to view memory, protection might be bypassed |
| $P_9$ | An attacker might still be able to executed code using the return-into-libc technique |
| $P_{10}$ | The protection can be bypassed by overwriting the exception handlers and causing an exception |
| $P_{11}$ | Overwriting memory marked as appropriate is still possible |
| $P_{12}$ | Only memory allocated with malloc is protected |
| $P_{13}$ | Current implementation might not be sufficient to stop injected code from being executed |
| $P_{14}$ | The heap is not randomized, code could still be injected there |
| $P_{15}$ | Mimicry attacks could be used to execute injected code |
| $P_{16}$ | Applications that need many privileges to execute do not benefit much |
| $P_{17}$ | False negatives are possible |
| $P_{18}$ | Only protects string copies when explicitly used |
| $P_{19}$ | Some impossible sequences of system calls are accepted as valid |
| $P_{20}$ | Possible changes to programs are not tracked, only definite changes |

Table 3: Protection limitation categories

| Code | Type category |
|---|---|
| $T_1$ | Detection: exploitation attempts are detected |
| $T_2$ | Prevention: The vulnerability is prevented |
| $T_3$ | Mitigation: operations an attacker can perform are limited or exploitation is made harder |

Table 4: Type categories

| Code | Response category |
|---|---|
| $R_1$ | Application is terminated |
| $R_2$ | Application crashes |
| $R_3$ | Application's state is restored and execution continues |
| $R_4$ | An exception is generated |
| $R_5$ | Unsafe operation is denied |
| $R_6$ | A problem is reported |

Table 5: Response categories

## 4.2 Deployment Tools

The tables 6, 7, 8 and 9 contain a list of the countermeasures that can be used at deployment time to protect an application against the vulnerabilities described in section 2. Due to space limitations, only the first author, the name of the countermeasure and the references to it are mentioned in this column. The coverage field indicates which types of vulnerability countermeasure offers protection for. The meaning of the codes used to indicate which vulnerabilities are covered by a countermeasure can be found in table 1. 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 applicability limitation column lists the most important limitations when applying the countermeasures (i.e. are there places where it can not be applied) that the countermeasure has, the meaning of the codes used here can be found in 2. The protection limitations column contains a list of the known limitations to the protection offered by a specific countermeasure. The meaning of the codes used in this column are in 3. The type column lists the type of countermeasure that is offered. Currently we have identified three major types of countermeasures: preventing countermeasures, detecting countermeasures and mitigating countermeasures, a mapping of the codes used in the column to the actual types is in table 4. The final column contains the response the countermeasure will have when trying to prevent a vulnerability or when detecting or mitigating an exploitation attempt. The codes for this column are in table 5.

| Countermeasure | Coverage | Comp. Cost | Mem. Cost | App. lim. | Prot. lim. | Type | Response |
|---|---|---|---|---|---|---|---|
| **Bounds Checking** | | | | | | | |
| Kendall (Bcc) [65] | $V_1+V_2+V_3+V_5$ | High | High | $A_1$ | $P_1$ | $T_1+T_2$ | $R_1$ |
| Steffen (RTCC) [119] | $V_1+V_2+V_5$ | High | High | $A_1+A_2$ | No | $T_1+T_2$ | $R_1$ |
| Austin (Safe C) [7] | $V_1+V_2+V_3+V_5$ | High | High | $A_1+A_3$ | No | $T_1+T_2$ | $R_1$ |
| Jones [62] | $V_1+V_2+V_5$ | High | High | $A_1+A_4$ | $P_2$ | $T_1+T_2$ | $R_1$ |
| Suffield [121] | $V_1+V_2+V_5$ | Very High | Very High | $A_1+A_4$ | $P_3$ | $T_1+T_2$ | $R_1$ |
| Lhee [78] | $V_1+V_2+V_5$ | Low-Medium | High | $A_1$ | $P_4+P_5$ | $T_1+T_2$ | $R_1$ |
| Oiwa (Fail-Safe C) [86] | $V_1+V_2+V_3+V_4+V_5$ | High | High | $A_1+A_4$ | No | $T_1+T_2$ | $R_1$ |
| Ruwase (CRED) [101] | $V_1+V_2+V_5$ | High | High | $A_1$ | $P_6$ | $T_1+T_2$ | $R_1$ |

Table 6: Deployment tools

| Countermeasure | Coverage | Comp. Cost | Mem. Cost | App. lim. | Prot. lim. | Type | Response |
|---|---|---|---|---|---|---|---|
| **Protection of Stackframes** | | | | | | | |
| Cowan (StackGuard) [27, 29] | $V_1$ | Low | Low | $A_1$ | $P_7 + P_8$ | $T_1$ | $R_1$ |
| Vendicator (Stack Shield ret stack) [123] | $V_1$ | Low | Low | $A_1$ | $P_7$ | $T_1$ | $R_1$ or $R_3$ |
| Vendicator (Stack Shield ret range) [123] | $V_1$ | Very Low | Very Low | $A_1$ | $P_9$ | $T_1$ | $R_1$ |
| Chiueh (Minezone RAD) [21] | $V_1$ | Low | Medium | $A_1$ | $P_7$ | $T_1$ | $R_1$ or $R_2$ |
| Chiueh (Read-only RAD) [21] | $V_1$ | Medium | Medium | $A_1$ | $P_7$ | $T_1$ | $R_1$ or $R_2$ |
| Etoh (Propolice) [39] | $V_1$ | Low | Low | $A_1$ | $P_8$ | $T_1$ | $R_1$ |
| Bray (.NET Stack Prot.) [17, 54] | $V_1$ | Low | Low | $A_1$ | $P_{10}$ | $T_1$ | $R_4$ |
| Xu (Split stack) [137] | $V_1$ | Very low | Very Low | $A_5$ | $P_7$ | $T_1$ | $R_1$ |
| Xu (SRAS) [137] | $V_1$ | Very low | Very Low | $A_5$ | $P_7$ | $T_1$ | $R_1$ |
| Özdoğanoğlu (SRAS) [90] | $V_1$ | Very low | Very Low | $A_5$ | $P_7$ | $T_1$ | $R_1$ |
| Lee (SRAS) [76] | $V_1$ | Very low | Very Low | $A_5$ | $P_7$ | $T_1$ | $R_1$ |
| Frantzen (StackGhost) [49] | $V_1$ | Low | Low | $A_6 + A_7$ | $P_7$ | $T_1$ | $R_1$ |
| Baratloo (Libsafe) [8] | $V_1$ | Low | Low | $A_9 + A_{10}$ | $P_7$ | $T_1$ | $R_1$ |
| Baratloo Libverify [8] | $V_1$ | Medium | Low | $A_9$ | $P_7$ | $T_1$ | $R_1$ |
| Snarskii (integrity check) [114] | $V_1$ | Low | Low | $A_9 + A_{10}$ | $P_7$ | $T_1$ | $R_1$ |
| Snarskii (Libparanoia) [113] | $V_1$ | Low | Low | $A_9 + A_{10}$ | $P_7$ | $T_1$ | $R_1$ |
| **Protection of Pointers** | | | | | | | |
| Cowan (PointGuard) [28] | $V_1+V_2+V_3+V_4+V_5$ | Medium | Low | $A_1$ | $P_8$ | $T_1$ | $R_1$ |
| Yong (Sec. Enf. Tool) [138] | $V_1+V_2+V_3$ | Medium - High | High | $A_1$ | $P_{11}$ | $T_1 + T_2$ | $R_1$ |

Table 7: Deployment tools continued

| Countermeasure | Coverage | Comp. Cost | Mem. Cost | App. lim. | Prot. lim. | Type | Response |
|---|---|---|---|---|---|---|---|
| **Protection of dynamically allocated memory** | | | | | | | |
| Robertson [99] | $V_2$ | Low | Low | $A_9$ | $P_{12}$ | $T_1$ | $R_1$ |
| Krennmair (Contrapolice) [70] | $V_2$ | Low | Low | $A_9$ | $P_5+P_{12}$ | $T_1$ | $R_1$ |
| Fetzer (HEALERS) [42, 44, 43, 45] | $V_2$ | High | High | $A_9$ | $P_5$ | $T_1$ | $R_1$ |
| Fault Isolation | | | | | | | |
| Wahbe (SFI) [128] | $V_1+V_2+V_3+V_4+V_5$ | Low | Low | No | $P_{13}$ | $T_3$ | $R_5$ |
| Small (MiSFIT) [111] | $V_1+V_2+V_3+V_4+V_5$ | High | Low | $A_7$ | $P_{13}$ | $T_3$ | $R_5$ |
| **Policy Enforcement** | | | | | | | |
| Erlingsson (SASI) [38] | $V_1+V_2+V_3+V_4+V_5$ | High | Low | No | $P_{13}$ | $T_3$ | $R_5$ |
| Evans (Naccio) [41] | $V_1+V_2+V_3+V_4+V_5$ | High | Low | No | $P_{13}$ | $T_3$ | $R_5$ |
| Goldberg (Janus) [53] | $V_1+V_2+V_3+V_4+V_5$ | Very Low | None | $A_{13}$ | $P_{15}+P_{16}$ | $T_3$ | $R_5$ |
| Provos (Systrace) [95] | $V_1+V_2+V_3+V_4+V_5$ | Low-Medium | None | $A_6$ | $P_{15}+P_{16}$ | $T_3$ | $R_5$ |
| Prevelakis (FMAC) [94] | $V_1+V_2+V_3+V_4+V_5$ | Very Low | None | No | $P_{15}+P_{16}$ | $T_3$ | $R_5$ |
| Kiriansky (Program Shepherding) [67] | $V_1+V_2+V_3+V_4+V_5$ | High | High | $A_{14}$ | No | $T_3$ | $R_5$ |
| **Anomaly detection** | | | | | | | |
| Forrest [47] | $V_1+V_2+V_3+V_4+V_5$ | High | High | No | $P_{15}+P_{16}+P_{19}$ | $T_3$ | $R_6$ |
| Sekar [107] | $V_1+V_2+V_3+V_4+V_5$ | High | Medium | No | $P_{15}$ | $T_3$ | $R_6$ |
| Wagner (NDFA) [126] | $V_1+V_2+V_3+V_4+V_5$ | Very High | Medium | $A_{14}$ | $P_{15}$ | $T_3$ | $R_6$ |
| Wagner (NDPDA) [126] | $V_1+V_2+V_3+V_4+V_5$ | Very High | Medium | $A_{14}$ | $P_{15}$ | $T_3$ | $R_6$ |

Table 8: Deployment tools continued

| Countermeasure | Coverage | Comp. Cost | Mem. Cost | App. lim. | Prot. lim. | Type | Re-sponse |
|---|---|---|---|---|---|---|---|
| **Non-executable Memory** | | | | | | | |
| "Solar Designer" (Non-exec. Stack) [116] | 1 | None | None | $A_6$ + $A_{11}$ | $P_9$ | $T_3$ | $R_1$ |
| "The PaX Team" (PAGEEXEC) [122] | $V_1+V_2+V_3+V_4+V_5$ | High | None | $A_6$ + $A_{11}$ | + $P_9$ | $T_3$ | $R_1$ |
| **Randomization** | | | | | | | |
| Barrantes (RISE) [9] | $V_1+V_2+V_3+V_4+V_5$ | High | High | No | $P_8$ | $T_3$ | $R_2$ |
| Kc [64] | $V_1+V_2+V_3+V_4+V_5$ | Low | High | $A_5$ | $P_8$ | $T_3$ | $R_2$ |
| "The PaX Team" (ASLR) [122] | $V_1+V_2+V_3+V_4+V_5$ | Very Low | Low | $A_6+A_8$ | $P_8$ | $T_3$ | $R_2$ |
| Xu (TRR) [136] | $V_1+V_2+V_3+V_4+V_5$ | Low | Medium | $A_{12}$ | $P_8$ | $T_3$ | $R_2$ |
| Chew [20] | $V_1$ | Low | Low | $A_6$ | $P_8+P_{14}$ | $T_3$ | $R_2$ |
| Bhatkar [11] | $V_1+V_2+V_3+V_4+V_5$ | None | Low-Medium | No | $P_8$ | $T_3$ | $R_2$ |
| **Format String Countermeasures** | | | | | | | |
| Cowan (FormatGuard) [26] | $V_4$ | Very Low | None | $A_4$ | $P_5$ | $T_1$ + $T_2$ | $R_1$ |
| Robbins (Libformat) [98] | $V_4$ | Very Low | None | $A_9$ | $P_5$ | $T_1$ + $T_2$ | $R_1$ |

Table 9: Deployment tools continued

## 4.3 Development Tools

Tables 10 and 11 contain a list of the development tools that were examined in this document. The columns containing the computational and memory costs for dynamic analysis countermeasures and general development tools are measures as a comparison to running non-instrumented programs. As such the costs of these countermeasures can not be compared to the costs of static analysis. Costs for countermeasures that use static analysis are measured in running time of the analysis. The other columns are the same as the ones used in the tables in section 4.2.

| Countermeasure | Coverage | Comp. Cost | Mem. Cost | App. lim. | Prot. lim. | Type | Re-sponse |
|---|---|---|---|---|---|---|---|
| **Static countermeasures** | | | | | | | |
| Larochelle (Splint) [40, 72] | $V_1+V_2+V_4$ | Low | Low | $A_1 + A_{16}$ | $P_{17}$ | $T_2$ | $R_6$ |
| Dor (CSSV) [34, 35] | | High | Very High | $A_1 + A_{16} + A_{18}$ | $P_{20}$ | $T_2$ | $R_6$ |
| Shankar [108] | $V_4$ | Low | Low | $A_1 + A_{16}$ | $P_{17}$ | $T_2$ | $R_6$ |
| Ashcraft (Meta-compilation) [5] | $V_5$ | Un-known | Un-known | $A_1 + A_{17}$ | $P_{17}$ | $T_2$ | $R_6$ |
| Bush (PREfix) [19, 74] | $V_1+V_2+V_3$ | Very High | High | $A_1$ | $P_{17}$ | $T_2$ | $R_6$ |
| Pincus (PREfast) [74] | $V_1+V_2+V_3$ | Low | Low | $A_1$ | $P_{17}$ | $T_2$ | $R_6$ |
| Xie (ARCHER) [135] | $V_1+V_2+V_5$ | High | | $A_1 + A_{19}$ | $P_{17}$ | $T_2$ | $R_6$ |
| Simon [109] | $V_1+V_2$ | Un-known | Un-known | $A_1 + A_{18}$ | No | $T_2$ | $R_6$ |
| Wagner (BOON) [127] | $V_1+V_2$ | Medium | Un-known | $A_1+A_4$ | $P_{17}$ | $T_2$ | $R_6$ |
| Rugina [100] | $V_1+V_2$ | Un-known | Un-known | $A_1$ | No | $T_2$ | $R_6$ |
| Ganapathy [50] | $V_1+V_2$ | Un-known | Un-known | $A_1$ | $P_{17}$ | $T_2$ | $R_6$ |
| Viega (ITS4) [124] | $V_1+V_2+V_4$ | Very Low | Very Low | $A_1 + A_{10}$ | $P_{17}$ | $T_2$ | $R_6$ |
| Wheeler (FlawFinder) [130] | $V_1+V_2+V_4$ | Very Low | Very Low | $A_1 + A_{10}$ | $P_{17}$ | $T_2$ | $R_6$ |
| "Secure Software, Inc" (RATS) [106] | $V_1+V_2+V_4$ | Very Low | Very Low | $A_1 + A_{10}$ | $P_{17}$ | $T_2$ | $R_6$ |

Table 10: Development tools

| Countermeasure | Coverage | Comp. Cost | Mem. Cost | App. lim. | Prot. lim. | Type | Re-sponse |
|---|---|---|---|---|---|---|---|
| **Dynamic countermeasures** | | | | | | | |
| Haugh (STOBO) [57] | $V_1 + V_2$ | Un-known | Un-known | $A_1 + A_{10}$ | $P_{17}$ | $T_2$ | $R_6$ |
| Hastings (Purify) [56] | $V_2 + V_3$ | Very High | High | No | No | $T_2$ | $R_1$ |
| Larson (Muse) [73] | $V_1 + V_2 + V_5$ | Very High | Medium | $A_1$ | $P_{17}$ | $T_2$ | $R_6$ |
| Ghosh (FIST) [51, 52] | $V_1$ | Un-known | Un-known | $A_1$ | $P_{17}$ | $T_2$ | $R_6$ |
| Fink (Prop. test-ing) [46] | $V_1 + V_2 + V_3 + V_4 + V_5$ | Un-known | Un-known | $A_1 + A_2$ | $P_{17}$ | $T_2$ | $R_6$ |
| Perens (Electric fence) [91] | $V_2$ | High | High | $A_9$ | $P_{12}$ | $T_2$ | $R_1$ |
| **General development tools** | | | | | | | |
| Miller [84] | $V_1 + V_2$ | None | None | $A_1 + A_2$ | $P_{18}$ | $T_2$ | $R_5$ |
| Messier [81] | $V_1 + V_2 + V_4$ | Un-known | Low | $A_1 + A_2$ | $P_{18}$ | $T_2$ | $R_5$ |
| Jim (Cyclone) [61, 55] | $V_1 + V_2 + V_3 + V_4 + V_5$ | Medium | Medium-High | $A_1 + A_{15}$ | No | $T_1 + T_2$ | $R_1 or R_6$ |
| Necula (Ccured) [85, 22] | $V_1 + V_2 + V_3 + V_4 + V_5$ | High | High | $A_1 + A_4$ | No | $T_1 + T_2$ | $R_1 or R_6$ |
| Deline (Vault) [82, 74] | $V_1 + V_2 + V_3 + V_4 + V_5$ | Un-known | Un-known | $A_1 + A_{15}$ | No | $T_1 + T_2$ | $R_1 or R_6$ |
| Kowshik (Control-C) [32, 69] | $V_1 + V_2 + V_3 + V_4 + V_5$ | Un-known | Un-known | $A_1 + A_{15}$ | No | $T_2$ | $R_6$ |

Table 11: Development tools continued

# 5 Related work

Although a lot of work has been done in categorizing vulnerabilities [3, 6, 12, 13, 14, 15, 58, 71] with many different taxonomies being made, each with its own relative strengths and weaknesses, little work has been done in categorizing the available countermeasures to some important vulnerabilities. Cowan et al. [30] have examined some countermeasures to buffer overflows, they examine a set of defenses including bounds checking modifications (Compaq c-compiler, Jones&Kelly compiler) safe languages (Java and ML), debugging tools (Purify), library patches (Snarskii FreeBSD patch) and compiler modifications (StackGuard and an early idea for PointGuard). Most of these countermeasures are only mentioned briefly, while only StackGuard and PointGuard are examined more in depth.

Wilander has published two documents in this area, one relating to static countermeasures [132] and one describing dynamic ones [133]. Wilander's documents are limited to publicly available tools with most focus being on comparing existing production tools. In [132], he examines ITS4, Flawfinder, RATS, Splint and BOON for comparison while in [133] he compares StackGuard, Stack Shield, Propolice, Libsafe and Libverify. In these documents he examines how effective these countermeasures are at respectively finding vulnerabilities in source code and preventing a wide range of attacks.

Lhee and Chapin [79] performed an extensive survey on techniques for exploiting buffer overflows and format string vulnerabilities. While they focus mainly on exploitation techniques, they also provide an overview of some countermeasures designed to stop buffer overflows and format string vulnerabilities. One of our main goals is to provide a comprehensive survey of countermeasures for vulnerabilities that could allow code injection, as such our vulnerability section contains more vulnerabilities: dangling pointer references and integer errors are not considered in the Lhee and Chapin paper. Because of the difference in focus we also describe more countermeasures and provide a synthesis that allows the reader to weigh the advantages and disadvantages of using one specific countermeasure as opposed to using another more easily.

# 6 Conclusion & Future Work

A few important properties of the countermeasures that we have described in this document were be identified: does a countermeasure try to prevent the vulnerability entirely, does it detect an attack and take subsequent action or does it try to mitigate the abuse of an attack? Or is it a combination of these properties? Other important properties like if the countermeasure requires changes to the original program or not, the impact of a countermeasure on performance, its compatibility with existing code, does it need access to the source code of the program it is protecting, its intrusiveness (are there programs that will stop working when applying a specific countermeasure?), its limitations (can it be bypassed?) and its coverage (stack-based overflows, heap-based overflows, format string vulnerabilities, ...). These properties were used to make a table of the countermeasures we described in this document. And can be useful when determining what kind of countermeasure might be most suited for a particular application.

Another observation that we have made is that many of the countermeasures presented here take an ad-hoc approach to prevent or mitigate the impact of specific vulnerabilities. We propose designing a model that describes the key abstractions that the compiler relies on to generate a program. These key abstractions, e.g. a function call, a virtual function table, stack, ..., are the ones that are generally abused by an attacker to gain control over the program's execution-flow. Conventions exist [36, 88] that describe some of these abstractions. However they are designed for system implementation and do not focus on the issue of security. By modeling these abstractions onto a machine model and modeling vulnerabilities, attacks and countermeasures onto this model we can reason about them on a more abstract level. It also provides a higher-level platform to compare specific countermeasures. One of our future work tracks is thus to devise this machine model and model some of the existing and new countermeasures onto it.

# References


[1] *Linux Programmer's Manual, section 3, printf()*. (Cited on page 18.)

[2] JTC 1/SC 22/WG 14. Iso/iec 9899:1999: Programming languages – c. Technical report, International Organization for Standards, 1999. (Cited on pages 19 and 42.)

[3] Robert P. Abbott, Janet S. Chin, James. E. Donnelley, William L. Konigsford, Shigeru Tokubo, and Douglas A. Webb. Security analysis and enhancements of computer operating systems. Technical report, 1976. (Cited on page 67.)

[4] anonymous. Once upon a free(). *Phrack*, 57, 2001. (Cited on page 10.)

[5] Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 143–159, Berkeley, California, USA, May 2002. IEEE Computer Society, IEEE Press. (Cited on pages 26 and 65.)

[6] Taimur Aslam. A Taxonomy of Security Faults in the Unix Operating System. Master's thesis, Purdue University, 1995. (Cited on page 67.)

[7] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, Florida, U.S.A., June 1994. ACM. (Cited on pages 41 and 61.)

[8] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *USENIX 2000 Annual Technical Conference Proceedings*, pages 251–262, San Diego, California, U.S.A., June 2000. USENIX Association. (Cited on pages 53 and 62.)

[9] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanović, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 281–289, Washington, District of Columbia, U.S.A., October 2003. ACM. (Cited on pages 49 and 64.)

[10] BBP. BSD heap smashing. http://bbp.krukh.net/blabla/BSD-heap-smashing.txt, May 2003. (Cited on page 10.)

[11] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, District of Columbia, U.S.A., August 2003. USENIX Association. (Cited on pages 56 and 64.)

[12] Richard Bisbey II and Dennis Hollingsworth. Protection Analysis Project: Final Report. Technical report, Information Sciences Institute, University of Southern California, 1978. (Cited on page 67.)

[13] Matt Bishop. A Taxonomy of UNIX System and Network Vulnerabilities. Technical Report CSE-9510, Department of Computer Science, University of California at Davis, May 1995. (Cited on page 67.)

[14] Matt Bishop. Vulnerability analysis. In *Proceedings of Recent Advances in Intrusion Detection 1999*, pages 125–136, West Lafayette, Indiana, U.S.A., September 1999. (Cited on page 67.)

[15] Matt Bishop and Dave Bailey. A Critical Analysis of Vulnerability Taxonomies. Technical report, Department of Computer Science, University of California at Davis, 1996. (Cited on page 67.)

[16] blexim. Basic Integer Overflows. *Phrack*, 60, December 2002. (Cited on page 19.)

[17] Brandon Bray. Compiler Security Checks In Depth. http://msdn.microsoft.com/library/en-us/dv_vstechart/html/vctchCompilerSecurityChecksInDepth.asp, February 2002. (Cited on pages 46 and 62.)

[18] Bulba and Kil3r. Bypassing Stackguard and Stackshield. *Phrack*, 56, 2000. (Cited on pages 8, 45 and 46.)

[19] William Bush, Jonathan Pincus, and David Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, June 2000. ISSN: 0038-0644. (Cited on pages 3, 27 and 65.)

[20] Monica Chew and Dawn Song. Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002. (Cited on pages 50 and 64.)

[21] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 409–420, Phoenix, Arizona, USA, April 2001. IEEE Computer Society, IEEE Press. (Cited on pages 45 and 62.)

[22] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the Real World. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 232–244, San Diego, California, U.S.A., 2003. ACM. (Cited on pages 22 and 66.)

[23] Matt Conover. w00w00 on Heap Overflows. http://www.w00w00.org/files/articles/heaptut.txt, 1999. (Cited on page 10.)

[24] Patrick Couscot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, Tucson, Arizona, U.S.A., January 1978. ACM. (Cited on page 26.)

[25] Crispin Cowan. StackGuard Mechanism: Emsi's Vulnerability. http://www.immunix.org/StackGuard/emsi_vuln.html, November 1999. (Cited on pages 44 and 46.)

[26] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 191–200, Washington, District of Columbia, U.S.A., August 2001. USENIX Association. (Cited on pages 55 and 64.)

[27] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Eric Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Proceedings of Linux Expo 1999*, Raleigh, North Carolina, U.S.A., May 1999. (Cited on pages 44 and 62.)

[28] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, District of Columbia, U.S.A., August 2003. USENIX Association. (Cited on pages 47 and 62.)

[29] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, U.S.A., January 1998. USENIX Association. (Cited on pages 44 and 62.)

[30] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings of the DARPA Information Survivability Conference & Exposition*, volume 2, pages 119–129, Hilton Head, South Carolina, U.S.A., January 2000. (Cited on page 67.)

[31] Rober DeLine and Manuel Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, U.S.A., May 2001. ACM. (Cited on page 22.)

[32] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory Safety Without Runtime Checks or Garbage Collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 69–80, San Diego, California, U.S.A., June 2003. ACM. (Cited on pages 9, 23 and 66.)

[33] Igor Dobrovitski. Exploit for CVS double free() for Linux pserver. `http://seclists.org/lists/bugtraq/2003/Feb/0042.html`, February 2003. (Cited on page 14.)

[34] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cleanness Checking of String Manipulation in C Programs via Integer Analysis. In *Proceedings of the Eight International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212, Paris, France, July 2001. Springer-Verlag. (Cited on pages 25, 28 and 65.)

[35] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 155–167, San Diego, California, U.S.A., 2003. ACM. (Cited on pages 25 and 65.)

[36] Richard Earnshaw. Procedure Call Standard for the ARM Architecture. Technical report, ARM, October 2003. (Cited on page 68.)

[37] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990. (Cited on page 3.)

[38] Úlfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *Proceedings of the New Security Paradigm Workshop*, pages 87–95, Caledon Hills, Ontario, Canada, September 1999. ACM. (Cited on pages 35 and 63.)

[39] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Technical report, IBM Research Divison, Tokyo Research Laboratory, June 2000. (Cited on pages 46 and 62.)

[40] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, January-February 2002. (Cited on pages 25 and 65.)

[41] David Evans and Andrew Twyman. Flexible Policy-Directed Code Safety. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 32–45, Oakland, California, U.S.A., May 1999. IEEE Computer Society, IEEE Press. (Cited on pages 36 and 63.)

[42] Christof Fetzer and Zhen Xiao. Detecting Heap Smashing Attacks Through Fault Containment Wrappers. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS'01)*, pages 80–89, New Orleans, Lousiana, U.S.A., October 2001. IEEE Computer Society, IEEE Press. (Cited on pages 55 and 63.)

[43] Christof Fetzer and Zhen Xiao. A Flexible Generator Architecture for Improving Software Dependability. In *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, pages 102–113, Annapolis ,Maryland, U.S.A., November 2002. IEEE Computer Society, IEEE Press. (Cited on pages 55 and 63.)

[44] Christof Fetzer and Zhen Xiao. An Automated Approach to Increasing the Robustness of C Libraries. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pages 155–164, Washington, District of Columbia, U.S.A., July 2002. IEEE Computer Society, IEEE Press. (Cited on pages 55 and 63.)

[45] Christof Fetzer and Zhen Xiao. HEALERS: A Toolkit for Enhancing the Robustness and Security of Existing Applications. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*, pages 317–322, San Francisco, California, U.S.A., June 2003. IEEE Computer Society, IEEE Press. (Cited on pages 55 and 63.)

[46] George Fink and Matt Bishop. Property-Based Testing: A New Approach to Testing for Assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, July 1997. (Cited on pages 32 and 66.)

[47] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, California, U.S.A., May 1996. IEEE Computer Society, IEEE Press. (Cited on pages 38, 40 and 63.)

[48] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, U.S.A., May 1999. ACM. (Cited on page 26.)

[49] Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *Proceedings of the 10th USENIX Security Symposium*, pages 55–66, Washington, District of Columbia, U.S.A., August 2001. USENIX Association. (Cited on pages 52 and 62.)

[50] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In *Proceedings of the 10th ACM conference on Computer and Communication Security*, pages 345–354, Washington, District of Columbia, U.S.A., October 2003. ACM Press. (Cited on pages 29 and 65.)

[51] Anup K. Ghosh and Tom O'Connor. Analyzing Programs for Vulnerability to Buffer Overrun Attacks. In *Proceedings of the 21st NIST-NCSC National Information Systems Security Conference*, pages 274–382, October 1998. (Cited on pages 32 and 66.)

[52] Anup K. Ghosh, Tom O'Connor, and Garry McGraw. An Automated Approach for Identifying Potential Vulnerabilities in Software. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 104–114, Oakland, California, U.S.A., May 1998. IEEE Computer Society, IEEE Press. (Cited on pages 32 and 66.)

[53] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium*, pages 1–13, San Jose, California, U.S.A., July 1996. USENIX Association. (Cited on pages 36 and 63.)

[54] Richard Grimes. Preventing Buffer Overflows in C++. *Dr Dobb's Journal: Software Tools for the Professional Programmer*, 29(1):49–52, January 2004. (Cited on pages 46 and 62.)

[55] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-Based Memory Management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002. (Cited on pages 21 and 66.)

[56] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter '92 USENIX conference*, pages 125–136, San Francisco, California, U.S.A., January 1992. USENIX Association. (Cited on pages 27, 31 and 66.)

[57] Eric Haugh and Matt Bishop. Testing C Programs for Buffer Overflow Vulnerabilities. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS'03)*, San Diego, California, U.S.A., February 2003. Internet Society. (Cited on pages 31 and 66.)

[58] John D. Howard. *An Analysis Of Security Incidents On The Internet 1989-1995*. PhD thesis, Carnegie Mellon University., 1997. (Cited on page 67.)

[59] Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press, 2001. (Cited on page 3.)

[60] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 2001. Order Nr 245470. (Cited on page 5.)

[61] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, California, U.S.A., June 2002. USENIX Association. (Cited on pages 20 and 66.)

[62] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, number 009-02 in Linköping Electronic Articles in Computer and Information Science, pages 13–26, Linköping, Sweden, 1997. Linköping University Electronic Press. (Cited on pages 42, 43 and 61.)

[63] Michel Kaempf. Vudo - an object superstitiously believed to embody magical powers. *Phrack*, 57, 2001. (Cited on page 10.)

[64] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 272–280, Washington, District of Columbia, U.S.A., October 2003. ACM. (Cited on pages 49 and 64.)

[65] Samuel C. Kendall. Bcc: Runtime Checking for C Programs. In *Proceedings of the USENIX Summer 1983 Conference*, pages 5–16, Toronto, Ontario, Canada, July 1983. USENIX Association. (Cited on pages 41 and 61.)

[66] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall Software Series, second edition edition, 1988. (Cited on page 3.)

[67] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, U.S.A., August 2002. USENIX Association. (Cited on pages 37 and 63.)

[68] klog. The Frame Pointer Overwrite. *Phrack*, 5, 1999. (Cited on pages 7 and 46.)

[69] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring Code Safety Without Runtime Checks for Real-Time Control systems. (Cited on pages 9, 23 and 66.)

[70] Andreas Krennmair. ContraPolice: a libc Extension for Protecting Applications from Heap-Smashing Attacks. http://www.synflood.at/contrapolice/, November 2003. (Cited on pages 54 and 63.)

[71] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A Taxonomy of Computer Program Security Flaws, with Examples. Technical report, 1993. (Cited on page 67.)

[72] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, Washington, District of Columbia, U.S.A., August 2001. USENIX Association. (Cited on pages 25 and 65.)

[73] Eric Larson and Todd Austin. High Coverage Detection of Input-Related Security Faults. In *Proceedings of the 12th USENIX Security Symposium*, pages 121–136, Washington, District of Columbia, U.S.A., August 2003. USENIX Association. (Cited on pages 31 and 66.)

[74] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting Software. *IEEE Software*, 21(3):92–100, May-June 2004. (Cited on pages 3, 22, 27, 65 and 66.)

[75] Doug Lea and Wolfram Gloger. glibc-2.2.3/malloc/malloc.c. Comments in source code. (Cited on page 10.)

[76] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. Enlisting Hardware Architecture to Thwart Malicious Code Injection. In *Proceedings of the First International Conference on Security in Pervasive Computing*, volume 2802 of *Lecture Notes in Computer Science*, pages 237–252. Springer-Verlag, 2003. (Cited on pages 52 and 62.)

[77] John R. Levine. *Linkers and Loaders*. Morgan-Kauffman, October 1999. (Cited on page 44.)

[78] Kyung-Suk Lhee and Steve J. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–90, San Francisco, California, U.S.A., August 2002. USENIX Association. (Cited on pages 42 and 61.)

[79] Kyung-Suk Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, 33(5):423–460, April 2003. (Cited on page 67.)

[80] David Litchfield. Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server. http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf, September 2003. (Cited on page 46.)

[81] Matt Messier and John Viega. Safe c string library v1.0.2. http://www.zork.org/safestr, November 2003. (Cited on pages 56 and 66.)

[82] Microsoft. Vault: a programming language for reliable systems. http://research.microsoft.com/vault/. (Cited on pages 22 and 66.)

[83] Microsoft. Buffer Overrun In RPC Interface Could Allow Code Execution. http://www.microsoft.com/technet/security/bulletin/MS03-026.asp, July 2003. (Cited on page 3.)

[84] Todd C. Miller and Theo de Raadt. strlcpy and strlcat – consistent, safe string copy and concatenation. In *Proceedings of the 1999 USENIX Annual Technical Conference (FREENIX Track)*, pages 175–178, Monterey, California, U.S.A., June 1999. USENIX Association. (Cited on pages 56 and 66.)

[85] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, Oregon, U.S.A., January 2002. ACM. (Cited on pages 21 and 66.)

[86] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure: Progress Report. In *Proceedings of International Symposium on Software Security 2002*, pages 133–153, Tokyo, Japan, November 2002. (Cited on pages 43 and 61.)

[87] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 49, 1996. (Cited on page 6.)

[88] The Santa Cruz Operation and AT&T. System V Application Binary Interface. Technical report, March 1997. (Cited on page 68.)

[89] The Santa Cruz Operation and AT&T. System V Application Binary Interface: SPARC Processor Supplement. Technical report, 1997. (Cited on page 52.)

[90] Hilmi Özdoğanoğlu, T. N. Vijaykumar, Carla E. Brodley, Ankit Jalote, and Benjamin A. Kuperman. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. Technical Report TR-ECE 03-13, Purdue University, February 2004. (Cited on pages 52 and 62.)

[91] Bruce Perens. Electric fence 2.0.5. http://perens.com/FreeSoftware/. (Cited on pages 54 and 66.)

[92] Frank Piessens. A taxonomy of causes of software vulnerabilities in Internet software. In *Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 47–52, Annapolis, Maryland, U.S.A., November 2002. IEEE Computer Society, IEEE Press. (Cited on page 3.)

[93] Frank Piessens. Secure Software Engineering. Draft, November 2003. (Cited on page 3.)

[94] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing Applications. In *Proceedings of the 2001 USENIX Annual Technical Conference (FREENIX Track)*, Boston, Massachusetts, U.S.A., June 2001. USENIX Association. (Cited on pages 37 and 63.)

[95] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, Washington, District of Columbia, U.S.A., August 2003. USENIX Association. (Cited on pages 37 and 63.)

[96] Gerardo Richarte. Four different tricks to bypass StackShield and StackGuard protection. http://www2.corest.com/files/files/11/StackguardPaper.pdf, April-June 2002. (Cited on pages 44, 45 and 46.)

[97] rix. Smashing C++ VPTRs. *Phrack*, 56, 2000. (Cited on page 10.)

[98] Tim Robbins. Libformat. http://www.securityfocus.com/tools/1818, October 2001. (Cited on pages 55 and 64.)

[99] William Robertson, Christopher Kruegel, Darren Mutz, and Frederik Valeur. Run-time Detection of Heap-based Overflows. In *Proceedings of the 17th Large Installation Systems Administrators Conference*, pages 51–60, San Diego, California, U.S.A., October 2003. USENIX Association. (Cited on pages 13, 54 and 63.)

[100] Radu Rugina and Martin C. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, Vancouver, British Columbia, Canada, June 2000. ACM. (Cited on pages 29 and 65.)

[101] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, California, U.S.A., February 2004. Internet Society. (Cited on pages 43 and 61.)

[102] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975. (Cited on page 33.)

[103] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000. (Cited on page 35.)

[104] Fred B. Schneider. Least Privilege and More. *IEEE Security & Privacy*, 1(5):55–59, September-October 2003. (Cited on page 33.)

[105] scut. Exploiting format string vulnerabilities. http://www.team-teso.net/articles/formatstring/, 2001. (Cited on page 18.)

[106] Inc Secure Software. Rats website. http://www.securesw.com/download_rats.htm. (Cited on pages 30 and 65.)

[107] R. Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 144–155, Oakland, California, U.S.A., May 2001. IEEE Computer Society, IEEE Press. (Cited on pages 39 and 63.)

[108] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–218, Washington, District of Columbia, U.S.A., August 2001. USENIX Association. (Cited on pages 26 and 65.)

[109] Axel Simon and Andy King. Analyzing String Buffers in C. In H. Kirchner and C. Ringeissen, editors, *International Conference on Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 365–379. Springer, September 2002. (Cited on pages 28 and 65.)

[110] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, and Douglas W. Clark. Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms. In *Proceedings of 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 259–271, Dallas, Texas, U.S.A., November 1998. (Cited on page 51.)

[111] Christopher Small. A Tool For Constructing Safe Extensible C++ Systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, pages 175–184, Portland, Oregon, U.S.A., June 1997. USENIX Association. (Cited on pages 34 and 63.)

[112] Nathan P. Smith. Stack Smashing Vulnerabilities In The Unix Operating System. http://reality.sgi.com/nate/machines/security/nate-buffer.ps, 1997. (Cited on page 6.)

[113] Alexander Snarskii. Libparanoia. http://www.lexa.ru/snar/libparanoia/. (Cited on pages 54 and 62.)

[114] Alexander Snarskii. Freebsd libc stack integrity patch. ftp://ftp.lucky.net/pub/unix/local/libc-letter, February 1997. (Cited on pages 54 and 62.)

[115] Brian D. Snow. The Future Is Not Assured – But It Should Be. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 240–241, Oakland, California, U.S.A., May 1999. IEEE Computer Society, IEEE Press. (Cited on page 3.)

[116] Solar Designer. Non-executable stack patch. http://www.openwall.com. (Cited on pages 48 and 64.)

[117] Solar Designer. JPEG COM Marker Processing Vulnerability in Netscape Browsers. http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt, July 2000. (Cited on page 10.)

[118] Eugene H. Spafford. Crisis and Aftermath. *Communications of the ACM*, 32(6):678–687, June 1989. (Cited on page 3.)

[119] Joseph L. Steffen. Adding Run-Time Checking to the Portable C Compiler. *Software: Practice and Experience*, 22(4):305–316, April 1992. ISSN: 0038-0644. (Cited on pages 41 and 61.)

[120] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition edition, 1997. (Cited on page 3.)

[121] Andrew Suffield. Bounds Checking for C and C++. Technical report, Imperial College, 2003. (Cited on pages 42 and 61.)

[122] The PaX Team. Documentation for the PaX project. http://pageexec.virtualave.net/docs/. (Cited on pages 48, 50 and 64.)

[123] Vendicator. Documentation for Stack Shield. http://www.angelfire.com/sk/stackshield. (Cited on pages 44 and 62.)

[124] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *16th Annual Computer Security Applications Conference*, New Orleans, Louisiana, U.S.A., December 2000. (Cited on pages 3, 30 and 65.)

[125] John Viega and Gary McGraw. *Building Secure Software*. Addison-Wesley, 2002. (Cited on page 3.)

[126] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–168, Oakland, California, U.S.A., May 2001. IEEE Computer Society, IEEE Press. (Cited on pages 39 and 63.)

[127] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the 7th Networking and Distributed System Security Symposium 2000*, pages 3–17, San Diego, California, U.S.A., February 2000. (Cited on pages 28 and 65.)

[128] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 203–216, Asheville, North Carolina, U.S.A., December 1993. ACM. (Cited on pages 33, 34 and 63.)

[129] Charles F. Webb. Subroutine call/return stack. *IBM Technical Disclosure Bulletin*, 30(11):221–225, April 1988. (Cited on page 51.)

[130] David A. Wheeler. Flawfinder website. http://www.dwheeler.com/flawfinder/. (Cited on pages 30 and 65.)

[131] David A. Wheeler. *Secure Programming for Linux and Unix HOWTO*, 3.010 edition, march 2003. (Cited on page 3.)

[132] John Wilander and Mariam Kamkar. A Comparison of Publicly Available Tools for Static Intrusion Prevention. In *Proceedings of NORDSEC 2002: the 7th Nordic Workshop on Secure IT Systems*, Karlstad, Sweden, November 2002. (Cited on page 67.)

[133] John Wilander and Mariam Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, San Diego, California, U.S.A., February 2003. Internet Society. (Cited on page 67.)

[134] Rafal Wojtczuk. Defeating Solar Designer's Non-executable Stack Patch. http://www.insecure.org/sploits/non-executable.stack.problems.html, 1998. (Cited on page 48.)

[135] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. In *Proceedings of the 9th European Software Engineering Conference, held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 327–336, Helsinki, Finland, 2003. ACM Press. (Cited on pages 28 and 65.)

[136] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. In *22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269, Florence, Italy, October 2003. IEEE Computer Society, IEEE Press. (Cited on pages 50 and 64.)

[137] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and K. Iyer Ravishankar. Architecture Support for Defending Against Buffer Overflow Attacks. In *Second Workshop on Evaluating and Architecting System dependabilitY*, pages 55–62, San Jose, California, U.S.A., October 2002. (Cited on pages 51 and 62.)

[138] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages Pages: 307 – 316. ACM, ACM Press, September 2003. (Cited on pages 47 and 62.)

[139] Greta Yorsh and Nurit Dor. The Design of CoreC. Technical report, Tel-Aviv University, April 2003. (Cited on page 25.)

[140] Yves Younan. An overview of common programming security vulnerabilities and possible solutions. Master's thesis, Vrije Universiteit Brussel, 2003. (Cited on pages 5, 19, 44, 48 and 49.)

[141] Cliff Changchun Zou, Weibo Gong, and Don Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 138–147, Washington, District of Columbia, U.S.A., November 2002. ACM. (Cited on page 3.)