A Methodology for Designing Countermeasures Against Current and Future Code Injection Attacks

Yves Younan, Wouter Joosen, Frank Piessens
DistriNet, Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200a, B-3001 Leuven, Belgium
{yvesy,wouter,frank}@cs.kuleuven.ac.be

Abstract

This paper proposes a methodology to develop countermeasures against code injection attacks, and validates the methodology by working out a specific countermeasure. This methodology is based on modeling the execution environment of a program. Such a model is then used to build countermeasures. The paper justifies the need for a more structured approach to protect programs against code injection attacks: we examine advanced techniques for injecting code into C and C++ programs and we discuss state-of-the-art (often ad hoc) approaches that typically protect singular memory locations. We validate our methodology by building countermeasures that prevent attacks by protecting a broad variety of memory locations that may be used by attackers to perform code injections. The paper evaluates our approach and discusses ongoing and future work.

Keywords: Advanced exploitation techniques, buffer overflows, C, C++, code injection, countermeasures

1 Introduction

Standard stack-based buffer overflows, where an attacker overwrites the return address on the stack by writing outside the bounds of an array, have become a well-understood problem and many programmers are producing code that is more resilient towards buffer overflows or are applying countermeasures that make using standard exploitation techniques harder. As a result, attackers are turning to more sophisticated techniques (e.g. indirect attacks) and are using new vulnerabilities to inject code into a program. These trends demonstrate the need for a more structured approach when building attack prevention countermeasures.

The memory locations that are generally abused by attackers to gain control over the execution flow of a program usually contain the abstractions that the run-time environment relies on to execute the program. Therefore we should strive to protect the entire execution environment from attacks. In this paper we will discuss an approach to protect the run-time environment. We have defined a model of the run-time environment of the program for the Intel Architecture for 32-bit [18] with GNU/Linux as the operating system and the GNU Compiler Collection [17] as the compiler for the languages C and C++. This model was then used to assess which memory locations besides the traditional return address could be attacked. The advantage of using such models to design countermeasures is that a countermeasure designer can work at a higher level of abstraction which reduces the effort needed to define and evaluate a specific countermeasure. Such a model will also allow us to compare the effectiveness of countermeasures more easily, allowing one to select a countermeasure that better suits a particular context.

The rest of the paper is structured as follows: section 2 discusses the advanced techniques besides stack-based buffer overflows that are being used to attack programs. Section 3 describes the countermeasures we built using our methodology and describes the approach taken when building the machine model. Section 4 discusses our approach and describes our future work, while section 5 examines existing countermeasures. Finally, section 6 presents our conclusion.

2 Advanced exploitation techniques

This section covers some more advanced techniques used by attackers to inject code into an application. We describe them here in detail as they are important to demonstrate how they can be used to bypass countermeasures. This demonstrates the need for the more methodical approach that we describe in section 3. We have divided the attacks in different subcategories, we start by describing indirect attacks in section 2.1. These require an intermediate step to be exploited (e.g. overwriting a data pointer to a different memory location). In section 2.2 we discuss regular buffer overflows in the data and bss sections of memory, which also require an attack technique different from the one used for regular stack-based overflows. Finally, section 2.3 describes format string vulnerabilities and how they can be used by attackers to perform code injection attacks.

As we do for the machine model and countermeasures of section 3, we focus on the IA32 architecture with GNU/Linux as operating system and the GNU Compiler Collection as compiler for illustrating the attack techniques in this section. As such, all pointers and integer memory locations that are mentioned here are four bytes large.

2.1 Indirect attacks

Indirect attacks are attacks where the attackers do not or can not reach their objective immediately (i.e. gaining control over the execution-flow) but need an intermediate step to achieve their goals. This intermediate step usually manifests itself in the overwriting of the memory location that some pointer refers to with the target memory location. When the pointer is later dereferenced for writing, the target location will be overwritten. This type of attack can be further divided into several subcategories. In section 2.1.1 we describe indirect pointer overwriting, where a data pointer is modified by a buffer overflow and made to point to a different location. Section 2.1.2 discusses a closely related attack: a heap-based buffer overflow is used to overwrite the memory management information of the dynamic memory allocator, which in turn can be used to overwrite arbitrary memory locations. Section 2.1.3 describes another indirect attack on the dynamic memory allocator, when memory is deallocated multiple times, attackers could again overwrite the memory management information, resulting in the overwriting of arbitrary memory locations. Integer errors, discussed in section 2.1.4 are a different kind of vulnerability: they are not exploitable by themselves but could result in a buffer overflow.

2.1.1 Indirect Pointer Overwriting

If the return address on the stack is protected by a countermeasure (like StackGuard [13], which places a random value before the return address and on return checks if the random value is unchanged), an attacker still might be able to exploit a stack-based buffer overflow vulnerability by using indirect pointer overwriting [10]. The attacker overwrites a data pointer to which attacker-controlled data will be written (e.g., a copy of a user-inputted string) and makes it point to the target memory location. When the pointer is later dereferenced for writing, it will overwrite the target memory location. This technique is illustrated in Figure 1.

The overflow is used to overwrite a local variable of *f1* holding the pointer to *value1*. The pointer is changed to point to the return address instead of pointing to *value1* (see dotted line 1). If the pointer is then dereferenced and the value it points to is changed at some point in the function *f1* to a value specified by attackers, they can then 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 later be executed could be interesting for an attacker to overwrite.

2.1.2 Exploiting heap-based overflows

Heap memory is dynamically allocated at run-time by the application. As is the case with stack-based arrays, arrays contained on the heap can, in most implementations, be overflowed too. The technique for overflowing is similar except that the heap grows upwards in memory instead of downwards. In contrast to stack-based buffer overflows, no return addresses are stored in this segment of memory so an attacker must use other techniques to gain control of the execution-flow. An attacker could of course overwrite a function pointer or perform an indirect pointer overwrite on pointers stored in these memory regions, but these are not always available.

Overwriting the memory management information that is generally associated with a dynamically allocated chunk [2, 6, 22, 44] is a more general way of attempting to exploit a heap-based overflow.

We will demonstrate how these dynamic memory allocators can be attacked by focusing on a specific implementation of a dynamic memory allocator called *dlmalloc* [29]. While dlmalloc is used as a basis for the allocator in the GNU/Linux operating system, these techniques could also be applied to similar allocators used in other operating systems. We will describe *dlmalloc* briefly and will demonstrate 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, which vary in size as the various allocation routines (*malloc*, *free*, *realloc*, ...) 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 one larger free chunk. These free chunks are kept in a doubly linked list, sorted by size. When the memory allocator at a later

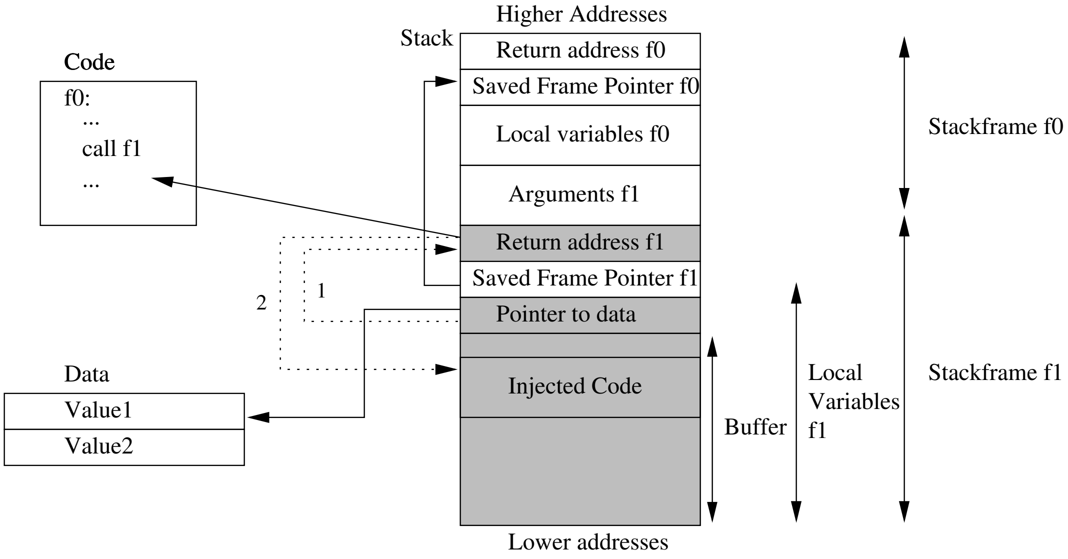


Figure 1. Stack-based buffer overflow using indirect pointer overwriting: full lines indicate normal state, dotted lines are changes due to the overwriting

time requests a chunk of the same size as one of these free chunks, the first chunk of that size in the list will be removed from the list and will be made available for use in the program (i.e. it will turn into an allocated chunk).

All memory management information (including this list of free chunks) is stored in-band. That is, 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 2 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] represents a free chunk that is located in a doubly linked list together with *chunk3* and *chunk4*. *Chunk3* is the first chunk in the chain: its backward pointer points to *chunk2* and its forward pointer points to a previous chunk in the list. *Chunk2* is the next chunk, with its forward pointer pointing to *chunk3* and its backward pointer pointing to *chunk4*. *Chunk4* is the last chunk in our example: its backward pointer points to a next chunk in the list and its forward pointer points to *chunk2*.

[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 *chunk2* is not entirely correct: if *chunk1* is in use, it will be used to store 'user data' for *chunk1* and not the size of *chunk1*. We have chosen to represent *chunk2* this way as this detail is not relevant to the discussion.

Figure 3 shows what could happen if an array that is located in *chunk1* is overflowed: an attacker has overwritten the management information of *chunk2*. The size fields are left unchanged (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 *chunk1* is subsequently freed, it will be coalesced together with chunk2 into a larger chunk. As *chunk2* 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 a 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 used to represent malloc chunks):

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

As a result, the value of the memory location that is twelve bytes after the location that *fd* points to will be overwritten with the value of *bk*, and the value of the memory location eight bytes after the location that *bk* points to will be overwritten with the value of *fd*. So in the example in Figure 3 the return address would be overwritten with a pointer to code that will jump over the place where *fd* will be stored

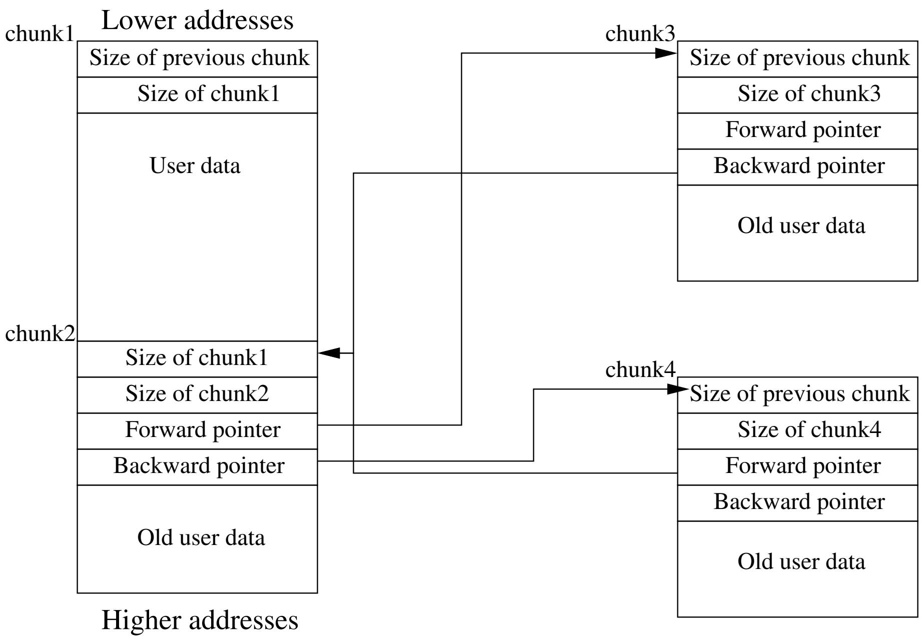


Figure 2. Heap containing used and free chunks

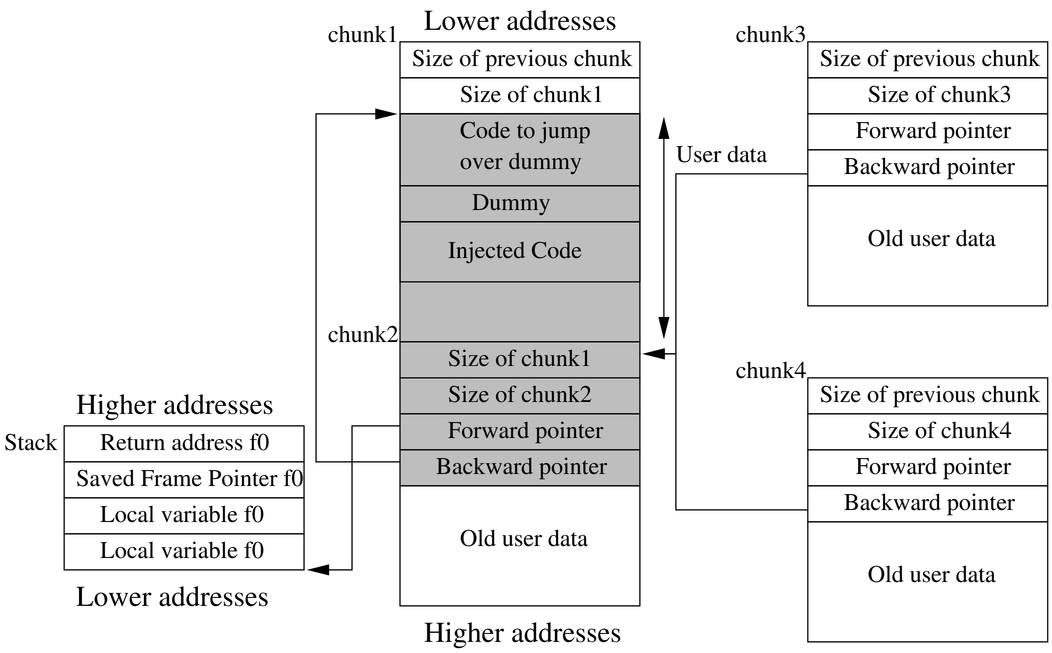


Figure 3. Heap-based buffer overflow

and will execute code that the attacker has injected. However, since the eight bytes after the memory that bk points to will be overwritten with a pointer to fd (illustrated as dummy in Figure 3), the attacker needs to insert code to jump over the first twelve bytes into the first eight bytes of his injected code. As with indirect pointer overwriting (see section 2.1.1), this technique can be used to overwrite arbitrary memory locations.

2.1.3 Exploiting dangling pointer references

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 [14].

We illustrate this using dlmalloc in Figure 4. The full lines in this figure are an example of what the list of free chunks of memory might look like when using the *dlmalloc* memory allocator. *Chunk1* is bigger than the *chunk2* and *chunk3* (which are both the same size), meaning that *chunk2* is the first chunk in the list of free chunks of equal size. When a new chunk of the same size as *chunk2* is freed, it is placed at the beginning of this list of chunks of the same size by modifying the backward pointer of *chunk1* and the forward pointer of *chunk2*.

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. As mentioned in the previous section: if a new chunk of the same size as *chunk2* is freed it will be placed before *chunk2* in the list. The following pseudo code demonstrates this (modified from the original version found in dlmalloc):

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

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

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
```

The forward and backward pointers of *chunk2* both point to itself. The dotted lines in Figure 4 illustrate what the list of free chunks looks like after a second free of *chunk2*.

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

But since both *chunk2−>fd* and *chunk2−>bk* point to *chunk2*, 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 (everything below 'size of chunk' in Figure 4) of the chunk for its own use.

Attackers can now use the same technique that we previously discussed to exploit the heap-based overflow (see Figure 3): 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 (or tries to free this one), it will again try to unlink *chunk2* which will overwrite the return address with the value of *chunk2's* backward pointer.

2.1.4 Exploiting integer errors

Integer errors 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 [21] 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 an unprepared program 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 do not usually lead to an exploitable condition.

Integer signedness errors on the other hand are more likely to occur and could lead to exploitable situations.

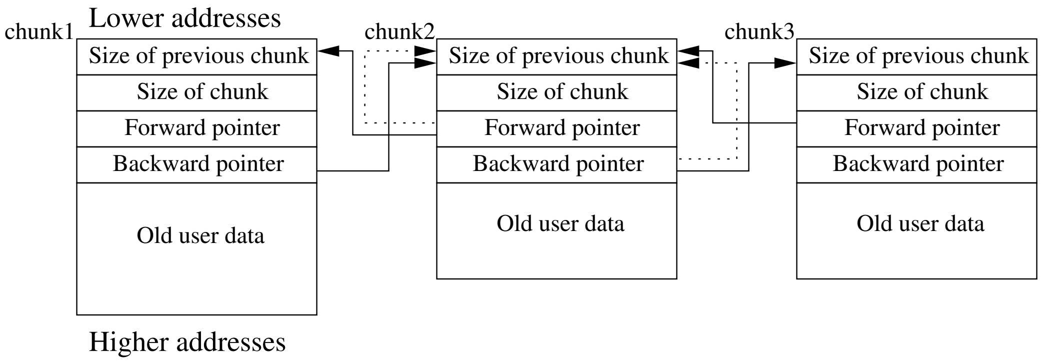


Figure 4. List of free chunks: full lines show a normal list of chunks, dotted lines show the changes after a double free has occurred.

When the programmer defines an integer, it is assumed to be a signed integer, unless explicitly declared unsigned. If this integer is later passed 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 stack or heap overflow if used in conjunction with a copy operation (e.g., *memcpy* or *memmove*[3] expects an unsigned integer as its size argument and when passed a negative signed integer, it will assume this is a large unsigned value).

More information about these attacks can be found in [7] and is also discussed extensively in the master thesis [51] of one of the authors.

2.2 Exploiting data and bss based overflows

Data memory contains global or static compile-time initialized memory and the *bss*[4] memory contains uninitialized global or static variables (that are initialized to 0 at load-time). These memory segments are constant in size: they will not grow during program execution. Overflows in these parts of memory are much the same as heap overflows: as no return addresses are present an attacker would normally either overwrite a function pointer or perform an indirect pointer overwrite.

Programs compiled with the GNU Compiler Collection can register functions as constructor and destructor functions. These functions will be executed respectively before and after the main function is executed. To know which functions to execute as constructor or destructor, a specific part of memory is reserved in which pointers to these functions are stored. These *ctors* and *dtors* sections respectively are stored after the *data* section and, hence, if a buffer located in this data section is overflowed it can be used to overwrite them. Note that since the *ctors* section has probably finished executing once an attacker is able to overflow a data-based buffer, this section is of less importance to an attacker. The layout of the header section between the *data* and *bss* sections of a statically linked application compiled with the GNU Compiler Collection (version 2.95.3) is as follows: *data*, *eh_frame*, *ctors*, *dtors*, *GOT* and *bss* (see Figure 5(a)). These sections will be mapped to memory in that order. Attackers can inject code by inserting their shellcode into the buffer they are overflowing in the data section and by continuing the overflow and overwriting a pointer in the *dtors* section to point to their code. When the program finishes the main function, it will call the injected code [36].

2.3 Attacks on format string vulnerabilities

Format functions are used to format the output of specific information. They have a variable amount of arguments and expect a format string as argument. 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. 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). 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].

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

[3] *memcpy* is the standard C library function that is used to copy memory from one location to another where memory areas may not overlap, *memmove* does the same but allows for overlapping memory areas.

[4] *bss* stands for "block started by symbol".

then be overwritten by crafting a special format string with %n specifiers [41]. Using this technique attackers can read and write arbitrary memory locations.

We described some advanced exploitation techniques in this section and focused on heap-based buffer overflows and dangling pointer references. We have focused on these two specific attacks as, next to the more global countermeasures that were designed using our machine model, we present the details of a countermeasure that would specifically make attacks on heap-based buffer overflows and dangling pointer references harder. However, the attacks describes described in this section are not the only way an attacker can perform code execution, [23, 40] describe a number of attacks on Multics where an attacker was able to gain higher privileges by misusing constructs specific to the implementation of Multics that were not a result of using a specific programming language. Such attacks could also be modeled when using a machine-model. This might make it easier for software engineers to fix these bugs.

3 Model-based countermeasure design

Most of the countermeasures described in section 5 use an ad hoc approach when trying to prevent vulnerabilities. In [52] we concluded that a more methodical approach is needed to combat code injection attacks. We propose doing this by building a model of the execution environment of the program based on the memory locations and abstractions that influence the execution flow. This model contains addresses and abstractions that can be used by an attacker to directly or indirectly influence the control flow of a particular application, supplemented with the locations that could lead to indirect overwriting of these addresses. Finally, these are supplemented with contextual information: what these specific memory locations are used for at different places of the execution flow and what operations are performed on them. This machine model allows a designer of countermeasures to view a platform in a more abstract way and as a result more effort can go into designing countermeasures rather than understanding obscure, possibly insignificant, platform details. It also allows a designer to take into account what the effects of a particular countermeasure are on a platform before having to implement it.

On most architectures code and data are loaded into separate segments of memory and have different properties (e.g. the code segment is typically read-only, the data segment is in some cases, on architectures that support it, non-executable). We can protect memory locations against code injection attacks by using a similar approach: by separating pointers and control-flow information from normal data like buffers, i.e. to store them in separate contiguous memory areas instead of storing them next to each other. Separating pointers and control-flow information would make it easier to add protections to these locations and would already prevent buffer overflow attacks from modifying them. We also suggest protecting the different memory sections by placing a non-writable page in between each section, making sure that a buffer overflow will not allow an attacker to write into other sections. A limitation of this approach is that, in its current form, it does not take attacks that can modify arbitrary memory locations (like format string vulnerabilities) into consideration. However, protecting against buffer overflow attacks that perform code injection is a first step that could afterwards be enhanced to protect against code injection attacks in general.

In the rest of the section we will illustrate our approach for the 32-bit Intel architecture (IA32), using the GNU/Linux operating system with gcc-2.95.3 for the language C.

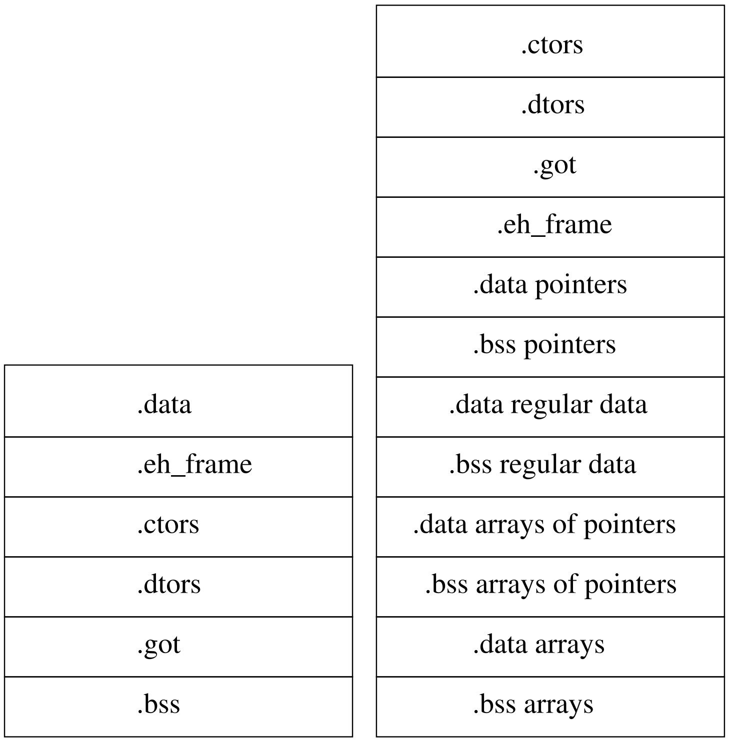
Separating control and data information requires several changes to the process memory of an application. We will describe the major changes here and will discuss one of these changes in more detail.

- Firstly, we must modify the way the stack is organized: The control data (e.g. the return address, the frame pointer, caller and callee save registers, pointers, ...) must be separated from the regular data. To do this we suggest making 3 stacks: one stack which contains the return addresses this is the regular stack and can still take advantage of the call and ret instructions. A second stack contains the frame pointers, local pointers and arrays of pointers[5]. Finally a third stack contains the other data.

- Secondly, dynamically allocated memory must have its memory management information stored out-band. To accomplish this its management information is stored at the beginning of the heap-section in a hashtable. The actual dynamically allocated memory simply contains the user-allocated memory.

- Finally, the memory in the data segment must be organized in a different order. The ctors and dtors sections would be stored first (and could, in theory, be placed in a read-only page), followed by the Global Offset Table (GOT), the exception handling frame, which are followed by pointers, regular data, arrays of pointers (again boundschecked) and finally normal arrays. Figure 5(a) shows the current layout and 5(b) shows the new layout.

[5] These arrays would be boundschecked: the impact on performance of this kind of boundschecking would be acceptable because arrays of pointers don't occur that often in a regular program



(a) Partial data segment layout (b) Modified data segment layout

Figure 5. Original and modified data segment layout

3.1 A partial machine model for the dynamic memory allocator

For the purposes of this paper we will focus on the part of the machine model dealing with dynamically allocated memory. As we described earlier the machine model will focus on memory locations that can be used by an attacker to modify the execution flow of an application. We represent the data structures and abstractions that are relied on during program execution using UML-class diagrams (see Figure 6). Specific data structures in memory are represented as classes, with the data members representing other data structures contained in this structure (e.g. the Heap contains MallocChunks). The order and frequency that particular parts of memory occur in, are denoted by the signs in front of the data member names:

+ denotes an ordered location.

- denotes that the order does not matter

* denotes that the part of memory can occur zero or more times, other data members occur exactly once.

For example, in Figure 6 the heap contains one or more malloc chunks in any order, with the malloc chunks containing exactly one instance of *prevsize* and *size* in order. The member functions represent operations that can be performed on specific memory locations, in the case of a chunk those are *allocate* and *free*. These member functions are redefined in their children if these operations are defined on these locations (e.g. a free chunk can be reallocated, but should not be freed a second time, hence *allocate* is redefined, while *free* is not).

To be able to represent the operations that are executed on these locations we have defined two primitives: *R(source)* and *W(source, destination)* which stand for read and write respectively. They can appear in forms where they respectively read or write to memory locations or registers. Besides these 2 primitives we need some control structures (like conditionals and loops) and temporary values (which are needed to temporarily store states in our model that indicate that a particular value was read into a variable at that point in time).

We have defined two types of representations to model an operation (see appendix A for an example of how the free operation would be expressed):

Define: defines what the operation does in sequential order in terms of memory location modifications (e.g. *Define MallocChunk.free(memory)* defines the *free* operation on *MallocChunk* memory which would expect a pointer to memory as argument, the details of this operation are in appendix A).

Modify: expresses what memory locations are modified by the function, expressed in terms of the supplied argument to the function or in terms of globally accessible data (e.g. *Modify MallocChunk.free(memory)* in appendix A specifies the modified memory functions for the *free* operation).

Applying the principles of control and data separation as described earlier to the heap results in the modification of the machine model depicted in Figure 7. The heap now contains one hashtable at the beginning of memory, followed by malloc chunks that contain the data of the chunk. The information contained in the *MallocChunk* in Figure 6 is now stored in the hashtable. When an *allocate* or *free* operation is performed, the required information is looked up in the hashtable. The countermeasure for preventing a double frees of dangling pointer reference is a fairly simple one: the unused bit in the size structure of Figure 6 is used to specify whether the current chunk is in use or not. This could also be solved by checking the *prev_inuse* bit of the next chunk, but the cost of accessing that bit is higher than using the unused bit in size, especially when using the extra indirection of the hashtable.

We are currently working on an implementation of this countermeasure to better assess the impact such a countermeasure has on performance and memory usage.

3.2 A machine model for the IA32-GNU/Linux-GCC-C platform

The machine model obviously includes more than the abstraction of the dynamic memory allocator. In this section we will describe the memory locations that form the full basis for our model, namely the memory locations that could be modified by attackers to gain control over the execution-flow of an application. These memory locations are specific for programs on a UNIX-like environment compiled with the GNU Compiler Collection, although many have equivalents in other operating systems and compilers.

Return address: The return address is stored on the stack and points to the memory location where execution must continue once a function has finished. This is the address that is usually attacked by stack-based buffer overflows and is also the memory location that most countermeasures protect.

Frame pointer: The frame pointer points to the location on the stack where a new stackframe was started and is used to reference local variables on the stack. The frame pointer is stored in a register and thus cannot be modified directly by an attacker. However when a new stack frame is started, the old frame pointer will be stored onto the stack and will be restored before

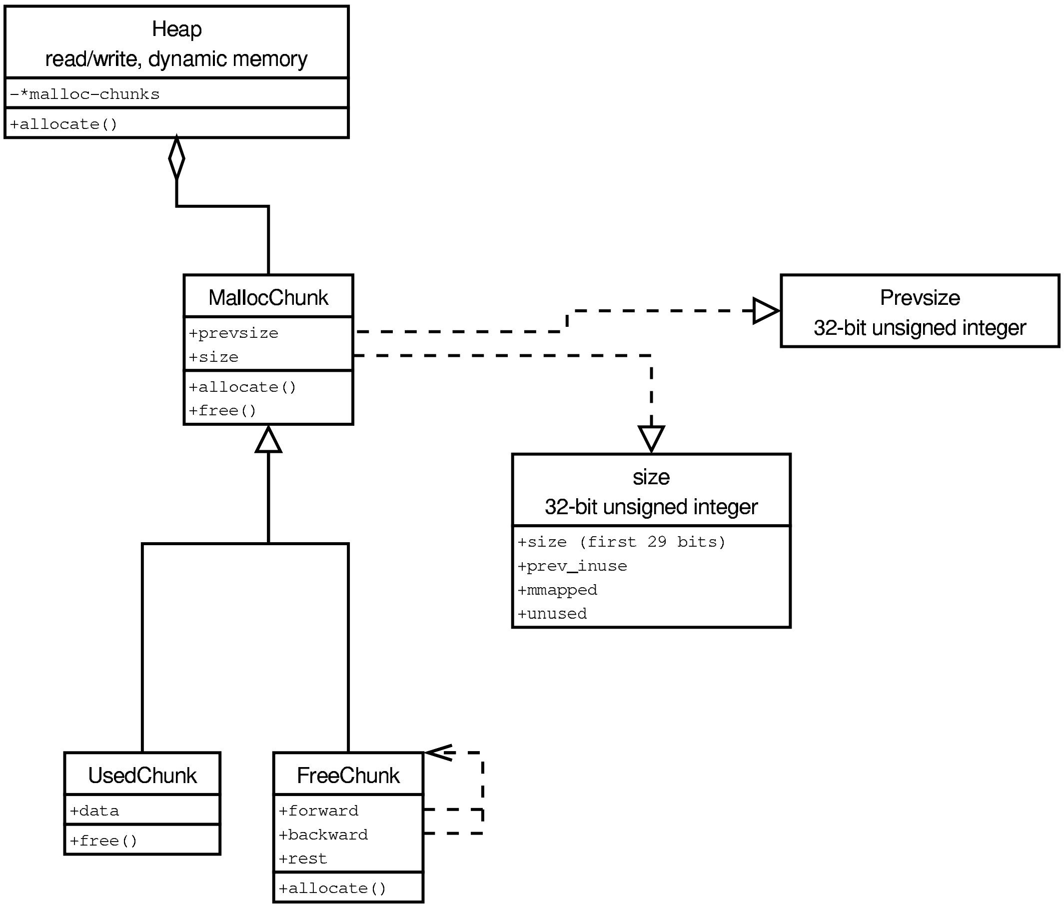


Figure 6. A typical machine model for the memory allocator.

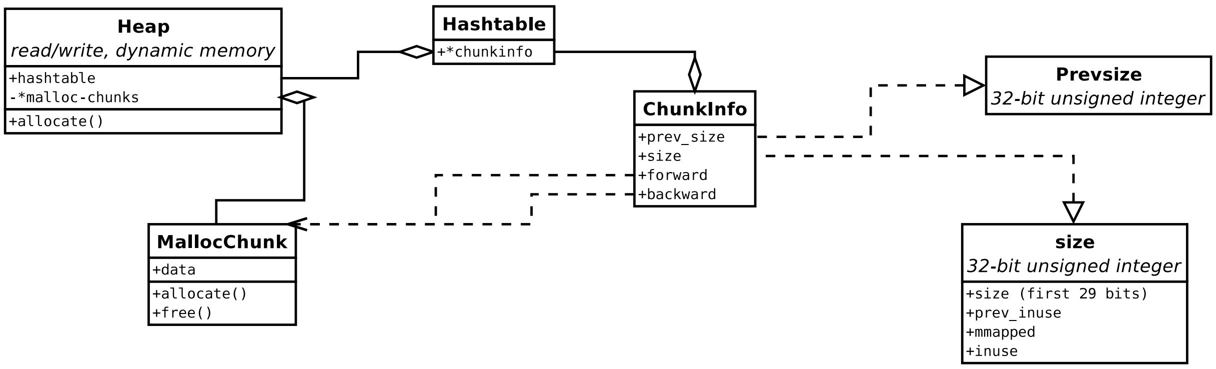


Figure 7. A modified machine model for the memory allocator.

returning from the function. When returning from a function the current frame pointer (in the register) will be copied to the stackframe pointer to free the stack and the stack-stored frame pointer will be copied to the frame pointer. The value that's at the top of the stack is now the return address and will be used to return from the function. If attackers modify the stack-stored frame pointer they will, at some point in the sequence of function calls, be able to influence where the return address is read from. Such an attack on this memory location has been described in detail in [26].

Function and data pointers: An easy way for an attacker to modify the control flow of a program is to modify a function pointer to point to injected code and to wait for the function pointer to be called. Data pointers can also be misused by attackers if they point to data that the attacker will be able to write to using that pointer. By modifying the contents of the data pointer and making it point to a new location, an attacker can subsequently modify the value stored at that memory location (indirect pointer overwriting, see section 2.1.1).

Virtual function pointers: Virtual function tables and pointers are used to support dynamic binding for C++. The binding of a member function declared as virtual then occurs at runtime based on the type of the object. To facilitate this dynamic binding the compiler adds a virtual table to every class that contains virtual functions. Then in each instance made of the class, a pointer (called the virtual pointer) is placed to this virtual table. Whenever a virtual member function is called, the virtual pointer is used to locate the virtual table and then the pointer at the appropriate virtual table slot is dereferenced and the method's code can be located. The virtual function pointer is stored in each object together with its data members. If attackers can overflow one of the buffers contained in the object they may be able to overwrite this pointer[6]. The attackers then make this pointer point to a dummy table that contains entries for the member functions of the object, that point to injected code. When one of these member functions is subsequently called, the injected code will be executed. More information on how these pointers can be attacked can be found in [37].

setjmp/longjmp buffer: The *setjmp* and *longjmp* functions are used to perform non-local returns in C, mainly for error handling. If a program wishes to return to a specific location in code when an error occurs, it can call *setjmp* with a pointer to a variable of type *jmp_buf* as argument. The *jmp_buf* type is a struct that contains information needed to restore the stack to its state at the moment of the *setjmp* call (i.e. removing stackframes of functions further down the call chain). The information stored in the *jmp_buf* are the callee save registers[7], the frame pointer register, the stack pointer register and the instruction pointer register (i.e. the *setjmp* return address). By restoring this information into the registers, the stack will be freed of lower level stack frames and execution will continue at the *setjmp* call site. If attackers modify the information (especially the saved instruction pointer register) in the *jmp_buf* and then causes an error that will make the program call a *longjmp*, they will be able to gain control over the execution flow.

Exception handling frame: For C++ programs that make use of exceptions, the compiler will generate an exception handling frame (called *eh_frame*) which could be overwritten to point to injected code. If an attacker can then force the program to generate an exception, the injected code would be executed.

dtors: The *dtors* section contains a list of pointers, terminated by a NULL, to functions that will be executed after the main function has finished. If an attacker overwrites one of these pointers[8], with a pointer to injected code it will be executed after the normal program code has finished executing.

Global Offset Table: The Global Offset Table (*GOT*) is used for dynamically linking code. It allows position independent code to access data at absolute virtual addresses. Instead of accessing the data directly this type of code references a position in its global offset table to retrieve the address, allowing the data to be stored at any memory location without breaking the code. An executable and every shared library each have their own *GOT*. The dynamic linker will calculate the absolute addresses of the requested symbols and will set the appropriate entries in the *GOT* to point to these addresses. The *GOT* also contains addresses of functions that have been dynamically loaded[9]. If an attacker overwrites one of the addresses with the location of injected code then the next time this function will be called, the attacker's code will be executed.

atexit: *atexit* is a C function defined in the ISO C99 standard that allows a program to register a function to be

[6] Or the virtual function pointer of the next object if the pointer is stored before the data, this is compiler dependent.

[7] These registers are supposed to have the same value after a function returns, so if a function wishes to use them it must save them and restore them before returning.

[8] If no destructor functions are registered the *dtors* section will still be present but will only contain the NULL terminator. An attacker can still exploit an empty dtors section by overwriting the NULL terminator

[9] The process of calling dynamically linked code is more complicated but not relevant to the discussion, see [30] for more details.

called when the program terminates normally (i.e. by explicitly or implicitly calling *exit*). The *atexit* function is passed a function pointer to the function that is to be registered and stores the function pointer in the *atexit* function list. When the program reaches the stage that *exit* is called, all functions in the function list will be executed one by one. If attackers overwrite one of these function pointers with a pointer to injected code, they will be able to execute arbitrary code [8].

Memory allocator information: The accounting information that is used to keep track of free chunks in a memory allocator can, as we demonstrated in section 2, be used to indirectly overwrite arbitrary memory locations if this information can be modified by a heap overflow or double free vulnerability.

Memory allocation hooks: Some implementations of *dlmalloc* allow a program to register hooks for the *malloc*, *realloc*, *free* and *memalign* function calls. These hooks contain pointers to functions that will be executed by these memory management functions whenever they are called. If an attacker overwrites one of these hooks with a pointer to injected code, the code will be executed whenever the respective memory allocation function is executed.

4 Discussion and future work

Using a modelbased approach to designing countermeasures has several advantages: the model allows one to reason about countermeasure design at a conceptual level and in a more systematic way. Because all relevant information is easily available and irrelevant information is not, the designer can more easily check for possible shortcomings in the proposed countermeasure. Many of the countermeasures in section 5 use an ad hoc approach that has led to many of these countermeasures being bypassed. These countermeasures could benefit from a more structured way of countermeasure design which the use of machine models offers. A further advantage of using a machine model is that it provides an easier way of comparing the effectiveness of countermeasures. Related to that, it would also offer a way of evaluating how two countermeasures could interfere with or complement each other.

Such a machine model is however strongly linked to the architecture, the operating system, the programming language and the compiler that it is based on. Because this dependency would limit the applicability of such a model, we are also in the process of designing a metamodel and devising a methodology for constructing machine models based on this metamodel.

The metamodel is an abstraction of several machine models: it provides uniformity when constructing machine models for a particular platform and allows one to work out the global principles of a particular countermeasure without having to deal with the implementation details. This allows countermeasure builders to design countermeasures at an even higher level of abstraction in a platform-independent way. By allowing platform-independent reasoning and by keeping the representation of machine models uniform, the metamodel simplifies the task of porting a countermeasure from one platform to another while being able to assess if a particular platform may need extra measures. For example, the Windows port [9] of StackGuard [13] neglected to take into account the way exceptions were handled on this platform and as a result attackers found a way of bypassing the countermeasure fairly quickly [32]. We argue that using the metamodel and machine models for the respective platforms would have made it easier to spot possible shortcomings when porting the countermeasure.

The methodology describes how to build a machine model for a particular platform based on this metamodel. It contains information that a particular expert of a platform (but not necessarily a security expert) must focus on to build a machine model for that platform. The machine model can then be used by a security engineer to build countermeasures to protect against code injection attacks. As a result the model also improves the possibility of collaboration: one person can build the machine model, another can design the countermeasure and yet another can implement the countermeasure.

A significant problem of the principle of control data and regular data separation that we use in our countermeasure is the possible impact on performance because of cache misses: if related data is stored in separate pages, multiple pages may have to be loaded into memory, to get the same information that would normally be stored in a single page. The performance impact of using separate stacks must also be investigated.

A shortcoming of the countermeasure for attacks against heap memory is the fact that we ignore the kind of data that is stored in the heap-allocated memory: objects (in the case of C++), structs and other pointers that may be stored in this memory could still lead to code injection attacks. Separating this data from regular data is something we plan to address in the near future.

5 Related work

Much work has gone into building countermeasures for the attacks described in section 2. In this section we will examine some of these countermeasures and will discuss their limitations. Many countermeasures have been developed and a more global overview can be found in a survey we recently completed [52]. We will focus on two types of countermeasures here: a subpart of preventative countermeasures (safe languages) and detecting countermeasures. Preventative countermeasures try to prevent a vulnerability from existing. Detecting countermeasures try to prevent a vulnerability from being exploited, which in most cases will lead to detection of an attempted exploitation, hence the name detecting countermeasures.

- Safe languages [19, 28, 33, 34] that are based on C or C++, offer a systematic way of solving the problems mentioned in section 2, by using a variety of techniques like managed memory, boundschecking, static checking, etc. The main disadvantage of these languages is that they change the language, so programs must be either explicitly written for that specific language or must be ported.

- Boundschecking solutions [3, 20, 25, 31, 35, 39, 45] solve the buffer overflow problem by ensuring that a pointer can not write outside the bounds of the object it is pointing to. This is done by instrumenting the program to check every pointer access. As a result the impact of these boundscheckers on performance is generally fairly high, limiting their use at deployment-time.

- Many countermeasures have been developed that will protect a single or multiple memory locations from exploitation. This can be done in a variety of ways: by placing a random value before the address being protected and making sure that the random value remains unchanged before using the memory location [13, 15, 27], by copying the memory location to a different area of memory and comparing the original to the copy before use [4, 11, 16, 47] or by calculating a checksum of several memory locations and encrypting (XOR) this with a random value and recalculating the checksum and encryption and comparing it to the original checksum before using the memory location [38]. Many of these countermeasures were designed ad hoc: they protect against a specific address being overwritten and can often be bypassed (especially using indirect pointer overwriting).

- Another approach taken by some countermeasures is to attempt to protect "all" memory locations: either by encrypting all pointers [12] or by enforcing a kind of access control on what memory locations pointers can reference [50]. These approaches are the most promising when trying to prevent code injection attacks. However if the program suffers from memory leaks or could be made to show the contents of specific memory locations (e.g. using a format string vulnerability), the countermeasure in [12] could be bypassed: attackers could guess the key by viewing the encrypted locations and encrypt the pointers to memory locations which they inject with the same key. The approach in [50] has some limitations: the slowdown is fairly large (but less than boundscheckers) and the static analysis that is used to determine what locations are appropriate for pointers to write to, may produce false negatives. This could result in memory locations that should not be written to, to be marked as writable.

- Marking memory as non-executable [42, 46] is also an approach that has been taken to prevent code injection attacks. However this approach has some limitations: memory that is not marked as non-executable but is still writable could still be used to perform these attacks (e.g. [42] only marks the stack as non-executable, the heap can still be used for code injection attacks). Another limitation is that these countermeasures can be bypassed by 'return-into-libc' attacks [43, 48], where attackers execute existing code (either code that is part of the program or library code) with arguments that they provide (e.g. they could call the libc wrapper for the *system* system call with a program they wish to run as argument).

- Another kind of countermeasure provides a randomized instruction set [5, 24]: instructions are encrypted (XOR) when they are stored in memory and decrypted before being loaded into the processor. Without access to the encryption key, code that attackers inject would be decrypted wrongly and the program would probably crash. This approach suffers from the same problem as [12]: if information is leaked, an attacker could guess the key and encrypt his instructions accordingly. Another major limitation of this countermeasure is that, unless hardware changes are made (e.g. a special chip), the impact on performance is extremely high as the code must be emulated.

- Attackers generally need to know where their code or the location they wish to overwrite is located in memory before they can perform a code injection attack. By randomizing the position at which specific memory starts [46, 49], it is harder for attackers to guess where their code is located or where specific vulnerable memory addresses are located. This countermeasure could also be bypassed if attackers can read out memory locations: by reading the position of some locations they could guess where their code or where the target memory location is located.

Some of these countermeasures offer good and complete protection, but can be impractical to use, either due to high performance penalties or because they require manual changes to the program or both. However, many of these countermeasure are designed in an ad hoc way and as a result suffer from limitations that could be misused by an attacker to bypass the countermeasure. Modifying these countermeasures to offer better protection against these attacks could be made easier if they were designed using the models that we described in section 3.

6 Conclusion

We have discussed how attackers can exploit vulnerabilities that were previously considered harmless and how they are using more advanced exploitation techniques to bypass countermeasures that aim to protect a single memory location. From this we concluded that there is a need for a more structured approach to counter attacks and we are building a model of the execution environment of a program to identify memory locations that may be used by an attacker to inject code.

Using machine models and a metamodel, we offer a higher level of abstraction when designing countermeasures which will allow designers to create countermeasures more easily and will allow them to detect problems sooner. The models also allow for easier collaboration when building countermeasures: the person building the model is not necessarily the person designing the countermeasure. Because the machine models are designed uniformly based on the metamodel, they also allow for easier porting of the countermeasure from one platform to another. Related to that, they also provide a platform for evaluating and comparing different countermeasures more easily.

The approach we present here is far from complete; we are in the process of constructing machine models for other, sufficiently different platforms. This establishes a firm basis for a metamodel that becomes a driver in refining our methodology.

References

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

[2] anonymous. Once upon a free(). *Phrack*, 57, 2001.

[3] T. M. Austin, S. E. Breach, and G. 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.

[4] A. Baratloo, N. Singh, and T. 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.

[5] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. 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., Oct. 2003. ACM.

[6] BBP. BSD heap smashing. `http://www.security-protocols.com/modules.php?name=News&file=article&si%d=1586`, May 2003.

[7] blexim. Basic integer overflows. *Phrack*, 60, Dec. 2002.

[8] P. Bouchareine. __atexit in memory bugs. Posted on the vuln-dev mailinglist `http://www.securityfocus.com/archive/82/151343`, Dec. 2000.

[9] B. Bray. Compiler security checks in depth. `http://msdn.microsoft.com/library/en-us/dv\_vstechart/html/vctchCompile%rSecurityChecksInDepth.asp`, Feb. 2002.

[10] Bulba and Kil3r. Bypassing Stackguard and stackshield. *Phrack*, 56, 2000.

[11] T. Chiueh and F.-H. 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, Apr. 2001. IEEE Computer Society, IEEE Press.

[12] C. Cowan, S. Beattie, J. Johansen, and P. 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., Aug. 2003. USENIX Association.

[13] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. 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., Jan. 1998. USENIX Association.

[14] I. Dobrovitski. Exploit for CVS double free() for linux pserver. `http://seclists.org/lists/bugtraq/2003/Feb/0042.html`, Feb. 2003.

[15] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. Technical report, IBM Research Divison, Tokyo Research Laboratory, June 2000.

[16] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, pages 55–66, Washington, District of Columbia, U.S.A., Aug. 2001. USENIX Association.

[17] Free Software Foundation. The gnu compiler collection. `http://gcc.gnu.org/`.

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

[19] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, California, U.S.A., June 2002. USENIX Association.

[20] R. W. M. Jones and P. 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.

[21] JTC 1/SC 22/WG 14. ISO/IEC 9899:1999: Programming languages – C. Technical report, International Organization for Standards, 1999.

[22] M. Kaempf. Vudo - an object superstitiously believed to embody magical powers. *Phrack*, 57, 2001.

[23] P. A. Karger and R. R. Schell. Multics security evaluation: Vulnerability analysis. Technical Report ESD-TR-74-193, HQ Electronic Systems Division, Hanscom Air Force Base, Massachusetts, U.S.A., June 1974.

[24] G. S. Kc, A. D. Keromytis, and V. 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., Oct. 2003. ACM.

[25] S. 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.

[26] klog. The frame pointer overwrite. *Phrack*, 55, 1999.

[27] A. Krennmair. ContraPolice: a libc extension for protecting applications from heap-smashing attacks. `http://www.synflood.at/contrapolice/`, Nov. 2003.

[28] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fähndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, May/June 2004.

[29] D. Lea and W. Gloger. glibc-2.2.3/malloc/malloc.c. Comments in source code.

[30] J. R. Levine. *Linkers and Loaders*. Morgan-Kauffman, Oct. 1999.

[31] K.-S. Lhee and S. 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., Aug. 2002. USENIX Association.

[32] D. Litchfield. Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server. `http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf`, Sept. 2003.

[33] Microsoft. Vault: a programming language for reliable systems. `http://research.microsoft.com/vault/`.

[34] G. Necula, S. McPeak, and W. 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., Jan. 2002. ACM.

[35] Y. Oiwa, T. Sekiguchi, E. Sumii, and A. 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, Nov. 2002.

[36] J. M. B. Rivas. Overwriting the .dtors section. Posted on the Bugtraq mailinglist `http://www.securityfocus.com/archive/1/150396`, Dec. 2000.

[37] rix. Smashing C++ VPTRs. *Phrack*, 56, 2000.

[38] W. Robertson, C. Kruegel, D. Mutz, and F. 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., Oct. 2003. USENIX Association.

[39] O. Ruwase and M. 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., Feb. 2004. Internet Society.

[40] J. H. Saltzer. Repaired security bugs in multics. In *Ancillary Reports: Kernel Design Project*, number MIT/LCS/TM-87, pages 1–4, Cambridge, Massachusetts, U.S.A., June 1977. Massachusetts Institute of Technology.

[41] scut. Exploiting format string vulnerabilities. `http://www.team-teso.net/articles/formatstring/`, 2001.

[42] Solar Designer. Non-executable stack patch. `http://www.openwall.com`.

[43] Solar Designer. Getting around non-executable stack (and fix). Posted on the Bugtraq mailinglist `http://www.securityfocus.com/archive/1/7480`, Aug. 1997.

[44] Solar Designer. JPEG COM marker processing vulnerability in netscape browsers. `http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt`, July 2000.

[45] J. L. Steffen. Adding run-time checking to the portable C compiler. *Software: Practice and Experience*, 22(4):305–316, Apr. 1992. ISSN: 0038-0644.

[46] The PaX Team. Documentation for the PaX project. `http://pageexec.virtualave.net/docs/`.

[47] Vendicator. Documentation for stackshield. http://www.angelfire.com/sk/stackshield.

[48] R. Wojtczuk. Defeating solar designer non-executable stack patch. Posted on the Bugtraq mailinglist `http://www.securityfocus.com/archive/1/8470`, Jan. 1998.

[49] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269, Florence, Italy, Oct. 2003. IEEE Computer Society, IEEE Press.

[50] S. H. Yong and S. 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 307–316. ACM, ACM Press, Sept. 2003.

[51] Y. Younan. An overview of common programming security vulnerabilities and possible solutions. Master's thesis, Vrije Universiteit Brussel, 2003.

[52] Y. Younan, W. Joosen, and F. Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.

A Representation of the *free* operation in the model

```
Modify MallocChunk.free(memory):
 malloc_state.maxfast.fastchunkbit
    -> R(malloc_state + 0) & 2
    -> R(malloc_state) & 2
 malloc_state.maxfast.anychunksbit
    -> R(malloc_state + 0) & 1
    -> R(malloc_state) & 1
 CHUNK.forward -> *(memory - 8 + 8)
    -> memory
 malloc_state.fastbins[(SIZE / 8) - 2]
    -> R(malloc_state + 4) + (((R(memory - 4) & 0xFFFFFFF8) / 8) - 2)
 PREVCHUNK.forward.backward
    -> R((memory - 8) - R((memory - 8) + 4) + 8) + 12
    -> R(memory - R(memory - 4)) + 12
 PREVCHUNK.backward.forward
    -> R((memory - 8) - R((memory - 8) + 4) + 12) + 8
    ->  R(memory - R(memory - 4) + 4) + 8
 CHUNK.forward.backward
    -> R((memory - 8) + 8) + 12 -> R(memory)+12
 CHUNK.backward.forward
    -> R((memory - 8) +12) + 8 -> R(memory + 4)+8
 NEXTCHUNK.size
    -> ((memory - 8) + R(memory - 8 + 4) + 4)
    -> ((memory - 8) + R(memory - 4) + 4)
 NEXTCHUNK.forward.backward
    -> R((memory - 8) + R(memory - 8 + 4) + 8) + 12
    -> R((memory - 8) + R(memory - 4) + 8) + 12
 NEXTCHUNK.backward.forward
    -> R((memory - 8) + R(memory - 8 + 4) + 12) + 8
    -> R((memory - 8) + R(memory - 4) + 12) + 8
 CHUNK.backward
    -> memory - 8 + 12
    -> memory + 4
 malloc_state.bins[0].forward.forward
    -> R(R(malloc_state + 48) + 8) + 8
 malloc_state.bins[0].forward.backward
    -> R(R(malloc_state + 48) + 8) + 12
 CHUNK.size.previnuse
    -> R(memory - 8 + 4) & 1
    -> R(memory - 4) & 1
 CHUNK.size.size
    -> R(memory - 8 + 4) & 0xFFFFFFF8
    -> R(memory - 4) & 0xFFFFFFF8
 malloc_state.top
    -> malloc_state + 40
```

```
Define MallocChunk.free(memory):
  if R(memory) = 0
    return
  CHUNK = R(memory) - 8
  SIZE = R(CHUNK.size.size)
  if SIZE <= 80
    W(malloc_state.maxfast.fastchunkbit, true)
    W(malloc_state.maxfast.anychunksbit, true)
    W(CHUNK.forward, R(malloc_state.fastbins[(SIZE / 8) - 2]))
    W(malloc_state.fastbins[(SIZE / 8) - 2], CHUNK)
  if R(chunk.size.mmap) = false
    W(malloc_state.maxfast.anychunksbit, true)
    NEXTCHUNK = CHUNK + SIZE
    NEXTSIZE = R(NEXTCHUNK.size.size)
    if R(CHUNK.size.previnuse) = false
      SIZE = SIZE + R(CHUNK.prev_size)
      CHUNK = CHUNK - R(CHUNK.prev_size)
      FD = R(CHUNK.forward)
      BK = R(CHUNK.backward)
      W(CHUNK.forward.backward, R(CHUNK.backward))
      W(BK.forward, FD)
    if NEXTCHUNK != R(malloc_state.top)
      NEXTINUSE = R((NEXTCHUNK + NEXTSIZE).size.previnuse)
      W(NEXTCHUNK.size, NEXTSIZE)
      if NEXTINUSE = false
        FD = R(NEXTCHUNK.forward)
        BK = R(NEXTCHUNK.backward)
        W(NEXTCHUNK.forward.backward, R(NEXTCHUNK.backward))
        W(BK.forward, FD)
        SIZE = SIZE + NEXTSIZE
      BK = R(malloc_state.bins[0])
      FD = R(malloc_state.bins[0].forward)
      W(CHUNK.backward, R(malloc_state.bins[0]))
      W(CHUNK.forward, FD)
      W(BK.forward, CHUNK)
      W(FD.backward, CHUNK)
      W(CHUNK.size.previnuse, true)
      W(CHUNK.size.size, SIZE)
      W((CHUNK + SIZE).prevsize, SIZE)
    else
      W(CHUNK.size.previnuse, true)
      W(CHUNK.size.size, SIZE + NEXTSIZE)
      W(malloc_state.top, CHUNK)
```