A Taxonomy of Buffer Overflow Characteristics

Matt Bishop, *Member*, *IEEE*, Sophie Engle, Damien Howard, *Member*, *IEEE*, and Sean Whalen

Abstract—Significant work on vulnerabilities focuses on buffer overflows, in which data exceeding the bounds of an array is loaded into the array. The loading continues past the array boundary, causing variables and state information located adjacent to the array to change. As the process is not programmed to check for these additional changes, the process acts incorrectly. The incorrect action often places the system in a nonsecure state. This work develops a taxonomy of buffer overflow vulnerabilities based upon characteristics, or preconditions that must hold for an exploitable buffer overflow to exist. We analyze several software and hardware countermeasures to validate the approach. We then discuss alternate approaches to ameliorating this vulnerability.

Index Terms—Protection mechanisms, software/program verification, security and privacy, arrays.

1 Introduction

Buffer overflows occur when a sequence of bytes of length n is placed into an array, or buffer, of length less than n. This simple error is all too common. In this paper, we focus on the buffer overflows that cause security problems.

It is well documented that buffer overflows can cause security problems. For example, a buffer overflow allowed a worm entry into a large number of UNIX systems in 1988 [1], [2], [3]. The trend has continued, with buffer overflows providing entry points for worms such as Blaster [4], Slammer [5], Apache/mod_ssl [6], and Code Red [7], [8]. Buffer overflows have also created other vulnerabilities in various programs and systems. For example, the Common Vulnerabilities and Exposures list [9] reported 100 buffer overflows identified in 2011 (so far), 413 identified in 2010, 587 identified in 2009, and 611 identified in 2008. The problem continues to exist despite efforts to eliminate it.

In this paper, we examine the causes for buffer overflow vulnerabilities by looking at the factors that create them. We present a classification scheme that distinguishes among the different types of buffer overflow vulnerabilities, and use this scheme to demonstrate the limits of proposed solutions. This suggests more effective solutions for handling multiple classes of buffer overflows. We do this by first deriving the preconditions necessary for the vulnerability to exist based on how these vulnerabilities are exploited, and then use pseudocode to refine these preconditions into common characteristics. We classify buffer overflows and their mitigations based on the characteristics each involves. Our objective is to show that classification based on these characteristics provides a systematic way to develop defenses for buffer overflows, and that in fact if any of the characteristics do not hold, the buffer overflow either does not exist or is not exploitable.

Numerous taxonomies of vulnerabilities [10], [11], [12], [13] present classifications based on general categories such as "unexpected change" or "logic error." This work expands upon Bishop's proposed classification for buffer overflows [14]. Cowan et al. [15] analyzed techniques for defending against buffer overflows; his scheme can be derived from the classification scheme we shall present. Other work in the specific vulnerability area of buffer overflow focuses on either attacks or defenses, and we will analyze it in the context of our classification scheme.

Section 2 of this paper provides some background illuminating key characteristics of the buffer overflow problem, as well as the principles underlying our classification scheme. The next sections present our classification of buffer overflows and analyzes it. We then examine proposed solutions in light of our classification. We conclude with some thoughts on future directions for the analysis and remediation (or elimination) of buffer overflow vulnerabilities.

2 Background

Architectural considerations are key to understanding buffer overflow attacks and defenses. The following discussion gives a high-level overview of such considerations and attacks. Aleph-One [16] and Conover [17] present detailed descriptions of how these attacks work.

2.1 Architectural Considerations

During program execution, buffer overflows can occur in three different areas of process memory: the data area, the stack, and the heap. The effects are constrained by the area in which the overflow occurs.

The data area of process memory provides space for nontransient variables such as global or static variables. These are defined before the process begins executing and are not deleted. They may or may not be initialized, but the memory for this area is typically contiguous. Variables in this area are bound to fixed virtual memory locations in the process address space.

- *M. Bishop is with the Department of Computer Science, University of California, One Shields Avenue, Davis, CA 95616-8592. E-mail: bishop@cs.ucdavis.edu.*
- *S. Engle is with the Department of Computer Science, University of San Francisco, 2130 Fulton Street, HR 533, San Francisco, CA 94117. E-mail: sjengle@cs.usfca.edu.*
- *D. Howard is with the Knobbe Martens Olson and Bear LLP, 14th Floor, 2040 Main Street, Irvine, CA 92614-3641. E-mail: Damien.Howard@kmob.com.*
- *S. Whalen is with the Department of Computer Science, Columbia University, 1214 Amsterdam Avenue, NY 10027. E-mail: shwhalen@gmail.com.*

Manuscript received 5 May 2011; revised 8 Dec. 2011; accepted 27 Dec. 2011; published online 11 Jan. 2012.
Recommended for acceptance by R. Sandhu.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2011-05-0129.
Digital Object Identifier no. 10.1109/TDSC.2012.10.

The stack contains data and variables that are allocated and deallocated as the process executes. The space allocated to the stack grows as functions are called because they add variables and other data to the stack, and shrinks as functions return because the variables and other data that functions allocated on the stack are released. On many systems, return addresses, processor status information, and call frame pointers are also placed on the stack. Variables on the stack are not bound to fixed virtual addresses. That is, the variable `x` in function `plugh` may have an address of 1,000 on the first call to `plugh`, and an address of 12,345 on the second call to `plugh`.

The heap is typically used for memory allocation done under the control of the program, for example by the process requesting that the system allocate memory for an array. When dynamically loadable modules are used, the modules are often loaded into the heap and then executed. Once assigned, memory in the heap remains bound to the variable until the process deallocates it.

The goals of buffer overflows are to change variables, return addresses, or function pointers. Variables and function pointers may be modified by overflows in any area, while return addresses, typically stored on the stack, may be modified only on the stack. Changing return addresses and function pointers alters the flow of control (including causing the program to crash), while changing variables results in changes to data. If the modified variable is used in a conditional expression later, changes to data may also alter the control flow. This suggests two broad classes of buffer overflow attacks.

2.2 Data Buffer Overflow

A data buffer overflow occurs when input overwrites existing data, causing the program to act in a manner that violates the (explicit or implicit) security policy. In terms of architecture, it requires that an array and a variable be allocated such that overflow from the array alters the contents of the variable. The variable controls some aspect of security-critical behavior.

An example of this is the apocryphal `login` program buffer overflow. In that program, the buffer holding the user-input password and the buffer holding the hashed value of the password were adjacent. The user-input buffer was 80 characters long. The program prompted the user for a login name, retrieved the hashed password corresponding to that login and stored it in the second buffer. The user was then prompted for the password. The user entered the password, which was then hashed and compared to the stored hash; a match authenticated the user. The vulnerability arose because the length of the password that the user entered was not checked. The attacker would pick any password of length 8, and generate the corresponding hash. The attacker would then enter the password, hit 72 spaces, and then type in the hash corresponding to the entered password. That overwrote the retrieved hash. Then, the program computed the hash of the entered password and compared that with the stored hash—which, having been supplied by the user, matched. This authenticated the user, regardless of whether the user knew the actual password corresponding to the account.

This is an example of a *direct* data buffer overflow. The data buffer overflow is called *indirect* if the value changed indirectly affects the selection or modification of a value that controls some aspect of security-critical behavior. Attacks that change pointers to refer to input data fall into this class.

2.3 Executable Buffer Overflow

An executable buffer overflow occurs when executable code is loaded into a buffer, and some quantity (a return address or function pointer) is altered to cause that code to be executed. Its simplest incarnation involves a buffer allocated on the stack. The data being entered is typically a set of machine-language instructions to be executed. The value in the location where the return address is stored is reset to be the address of the machine instructions in the buffer. As a result, when the routine returns, and the value in the location for the return addresses is popped and put into the Program Counter (PC), the input machine instructions execute.

A good example of this is the `fingerd` flaw that the Internet Worm of 1988 exploited [1], [3], [13]. That program used a library function to load input into a buffer on the stack. The library function did not check the length of the input. The buffer was 256 characters long, and was allocated by the caller. When the library function was called, the return address was pushed onto the stack beyond the end of the buffer. The input routine would load the characters into the buffer. By providing input of more than 256 bytes, the attacker could overflow the buffer and change the value stored in the location where the return address had been placed. On return, the new value would be the location where execution resumed. The attacker used this to execute a small program called the "grappling hook" that compiled and executed a second small program, which in turn pulled over components of the worm, linked them, and executed the worm.

Executable buffer overflows may not include the instructions to be executed in the buffer. Heap spraying attacks scatter segments of executable code throughout the heap [18]. Later, a buffer overflow attack can transfer control to one of those segments. Variants include the JIT spray [19] and heap feng shui [20]. If a buffer overflow changes the return address or a function pointer, then the result is an executable buffer overflow.

If the executable buffer overflow alters process state information, such as the return address or processor status word, then the overflow is direct, as in the above example. If it does not alter process state information, for example altering a function pointer only, then it is said to be indirect.

2.4 Format Strings and Buffer Overflows

Two aspects of strings that can cause buffer overflows are relevant to our analysis. The first is the related but distinct "format string" attack; the second is the internal conversion of input strings.

A format string attack occurs when an input string contains formatting commands. The input string itself does not overflow a buffer, but when it is applied to other data, the result does. As an example, suppose the array `buf` is allocated to have 100 characters. The following code is intended to print the number "139" into the `buf` array:

```
sprintf(buf, input_string, 139);
```

If `input_string` is "%d," no overflow will occur. But if `input_string` is "%d $c_1 \ldots c_{98}$," where $c_1 \ldots c_{98}$ are characters, a buffer overflow will occur. In essence, the semantics of the operation `sprintf` upon the input string causes the buffer overflow, rather than the input string itself.

A second instance of this type of expansion occurs when one inputs a string that is converted to a longer string. When an input string is converted to Unicode, the conversion adds several extra characters to enable the decoding engine to determine which characters were entered. For example, the "." character can be encoded as 2 bytes with the hex values `C0 AE`, or as 6 bytes with the hex values `FC 80 80 80 80 AE` in the UTF-8 scheme. The Basic Greek "Δ" character would be represented as `%u0394` in an alternate Unicode representation. The addition of the extra characters can cause the transformation of the input string to overflow the buffer (see for example Esser [21]).

Both these expansions take input strings that will not themselves overflow the buffer, and transform them into strings that will overflow the buffer. This adds a new dimension to the idea of an input string overflowing a buffer. More precisely, we require either that the input string overflow the buffer, or that it be transformed in such a way that it overflows the buffer based upon some property of the input string. Thus, both of the above examples qualify as input strings that cause buffer overflows.

However, format string attacks may, or may not, exploit buffer overflow vulnerabilities. Format string attacks may write data to arbitrary locations by using the "%n" formatting element. These attacks do not overflow buffers because they write data to specific memory locations [22]. Hence, we distinguish between format string vulnerabilities in general and buffer overflow vulnerabilities. Our work does not address format string vulnerabilities.

2.5 Summary

The difference between the types of buffer overflows lies in the use of the values in the locations beyond the buffer. Direct data buffer overflows change a variable value, and either a conditional is affected by the variable, or the variable's value is output, whereas indirect data buffer overflows change a pointer, causing the program to use incorrect data. Direct executable buffer overflows cause the flow of control to jump to a location other than that to which the program should have jumped, whereas indirect executable buffer overflows change a pointer, causing the execution of instructions that should not be executed at that point in time. The indirect cases are essentially the same as the direct cases, except that the instructions or data are already in the process memory.

If a buffer overflow causes the program to crash, either the program jumps to a nonexecutable location (executable) or tries to access inaccessible data (data). We do not consider this case further.

3 Preconditions

Before we are able to classify buffer overflow vulnerabilities, we must first identify the preconditions necessary for these vulnerabilities to exist. We identify these preconditions by examining how an attacker might exploit a buffer overflow vulnerability. We define these initial preconditions such that disabling any single precondition prevents the exploitation of that vulnerability.

As an example, let us consider a typical problem: a web server fails to check the length of a string read from the network. For our purposes, the policy of the site running the web server is that the web server may execute only a specified set of commands, and may only reveal the contents of the web pages it serves. The failure to check the bounds of the input string allows the attacker to supply an input string that corrupts the running web server process, causing it to violate the policy. Adding a check of the buffer length negates this precondition.

3.1 Executable Buffer Overflow

First, consider a buffer overflow on the stack. Here, the attacker inputs a string containing no-ops, a small machine-language program, and multiple copies of an address corresponding to one in the buffer before the machine language program. When this string is read and stored on the stack, it overwrites the return address. When the input routine returns, the machine-language program is executed.

Stated succinctly, this attack is: "input an extra long stream of instructions and a return address; the return address overwrites the one on the stack; on return, the corrupted address causes a return into the stack and executes the machine-language program stored there." Therefore, this attack has five parts:

1. Input a string that is longer than the array.
2. The string contains, or after transformation contains, both instructions and a return address.
3. The return address overwrites the one on the stack.
4. On return, the corrupted address causes a return into the stack.
5. Executes the machine language program stored there.

We transform each part of this attack into corresponding preconditions as follows:

- P1. The length of the (possibly transformed) input string is longer than that of the buffer.
- P2. The input (and possibly transformed) string contains instructions and/or addresses.
- P3. Input can change the stored return address without the change being countered.
- P4. The program can jump to memory in the stack.
- P5. The program can execute instructions stored in the stack.

Note that preconditions P2 and P5 seem redundant. They are not. One may be able to input data that contains valid instructions, but the process may not execute them due to Data Execution Prevention (DEP) mechanisms such as the NX bit [23, p. 143] of AMD or the XD bit [24, pp. 4-43] of Intel architectures.

Similarly, preconditions P4 and P5 are distinct. Precondition P4 states that the flow of control can transfer into an area reserved for data. Precondition P5 says instructions may be executed in that area. If precondition P4 is met and precondition P5 is not, the program will attempt to execute instructions, causing an exception.

Now, consider a buffer overflow that does not alter the return address. Here, the attacker either places instructions to be executed into the space of the process that is being attacked, or locates these instructions in the program. The placement may occur by supplying the instructions as input, as part of the environment, as command-line arguments, or through any other mechanism. Next, the attacker locates an array followed by a function pointer variable. The attacker overflows the array, and places the address of the instructions into the function pointer variable. At a later time, when the function pointer is invoked, the instructions are executed, compromising the system.

First, we assume that the instructions are resident in the heap of the process space. Then the attack takes several steps:

1. Input a string that is longer than the array.
2. The string contains, or after transformation contains, data and the address of instructions.
3. The address overwrites the function pointer.
4. On invocation, the corrupted pointer causes a jump to instructions in the heap.
5. The process can execute instructions stored in the heap.

This leads to the following set of preconditions:

P6. The length of the (possibly transformed) string is longer than that of the buffer.
P7. The input (and possibly transformed) string contains addresses.
P8. Input can change the value in the function pointer variable without being countered.
P9. The program can jump to the heap.
P10. The program can execute instructions in the heap.

Note the parallel between these indirect executable buffer overflow preconditions and the ones for the direct executable buffer overflow. In particular, if "return address" replaces "function pointer" and "stack" replaces "heap," the two are the same.

3.2 Data Buffer Overflow

Now, consider the data buffer overflow. This attack differs from the executable buffer overflow in that no new instructions are executed; data is merely changed, and as a result the process executes existing instructions that would otherwise not have been executed. Here, the attacker locates an array followed by some particular variable (which may be an array, as in the login example cited earlier). The attacker overflows the array, and as part of the overflow sets the variable to the desired value. The new value causes some unauthorized action to be taken.

This leads to the following steps:

1. Input a string that is longer than the array.
2. The string contains, or after transformation contains, data that matches the type of the variable to be changed.
3. The overflow data alters the particular variable's value.
4. The program reads that variable's value.
5. The altered value of the variable causes the program to execute instructions that the unaltered variable value would cause not to be executed.

The resulting preconditions are:

P1. The length of the (possibly transformed) string is longer than that of the buffer.
P2. The input (and possibly transformed) string contains data of the type of the particular variable.
P3. The value stored in the particular variable can be changed without being countered.
P4. The particular variable determines which execution path is to be taken at a future point in the execution of the process.

The differences between this set of preconditions and the previous two sets is instructive. The first three are essentially the same as in the other sets. All involve overflowing the buffer (precondition P11). All involve changing a memory location to contain a value that will be interpreted as a legitimate value (precondition P12). Legitimacy is important, because the value supplied must be one that the process *could* have placed there. Otherwise, the change may be detected (precondition P13). The last precondition generalizes the notion of executing input instructions. The instructions are not input, of course, but must still be executed. There is no question that the instructions *can* be executed because the program has them in one control flow path. This eliminates the need for a precondition stating that the instructions can be executed. The only question is whether the instructions leading to the compromise will be executed, and precondition P14 speaks to that.

An indirect data buffer overflow is similar to a direct data buffer overflow, the difference being that the particular variable being modified points to a value that determines which execution path is taken. Noting this, we can immediately state the preconditions:

P15. The length of the (possibly transformed) string is longer than that of the buffer.
P16. The input (and possibly transformed) string contains addresses.
P17. The address stored in the particular pointer variable can be changed without being countered.
P18. The value pointed to by the particular pointer variable determines which execution path is to be taken at a future point in the execution of the process.

4 CHARACTERISTICS

Many of the buffer overflow preconditions share similarities. For example, P1, P6, P11, and P15 all capture the overflowing of the buffer, but do so for different types of buffer overflow vulnerabilities. We refine these preconditions into a single *characteristic*, which captures preconditions at a higher level of abstraction using pseudocode.

We first observe that P1, P6, P11, and P15 are all described as: "length of the (possibly transformed) string is longer than that of the buffer." We collapse these preconditions into a single new characteristic len:buff. Let $\mathrm{len}()$ be a function that returns the length of an array, and let the variable *input* refer to the input string (posttransformation) and variable *buffer* refer to the destination buffer. The characteristic len:buff holds when

$$\text{len:buff} \Leftrightarrow \mathrm{len}(input) > \mathrm{len}(buffer).$$

Each type of buffer overflow vulnerability also contains a precondition based on the type of information that may be included in the input string. For example, precondition P2 is described as "the input (and possibly transformed) string may contain instructions and/or addresses" and preconditions P7 and P16 are described as "the input (and possibly transformed) string may contain addresses." We define two pseudofunctions to capture these preconditions: the function $\mathrm{type}()$ returns the data type of a particular variable, and the function $\mathrm{contains}()$ returns true when a string may contain a particular data type. Let the data types for variables $addr$ and $inst$ be a memory address and instruction, respectively. We define the characteristics con:addr and con:inst as follows:

$$\begin{aligned}\text{con:addr} &\Leftrightarrow \mathrm{contains}(input, \mathrm{type}(addr))\\ \text{con:inst} &\Leftrightarrow \mathrm{contains}(input, \mathrm{type}(inst)).\end{aligned}$$

These characteristics are not mutually exclusive. If the input string may contain both addresses and instructions, then both con:addr and con:inst hold. This allows us to capture precondition P2, which requires both data types.

We take a similar approach to capture precondition P16, described as "the input (and possibly transformed) string may contain data of the type of the particular variable." We define the variable $ctrlvar$ to represent the "particular variable" that affects the control flow of the process, and capture this precondition with the characteristic con:ctrl as follows:

$$\text{con:ctrl} \Leftrightarrow \mathrm{contains}(input, \mathrm{type}(ctrlvar)).$$

We also observe that each type of buffer overflow vulnerability involves some sort of modification. For example, precondition P3 involves modification of a stored return address, P8 involves modification of a function pointer, and both P13 and P17 involve direct or indirect modification of a control flow variable. To capture these preconditions, we define a function $\mathrm{modify}()$ to determine whether a variable or pointer may be modified without being countered. We again use the variable $ctrlvar$ to represent a variable that affects the control flow of the process, and use the pointer variable $ctrlptr$ to refer to a pointer to that control flow variable. We also add variables $retnadd$ to reference a return address and $funcptr$ to reference a function pointer. The following characteristics mod:radd, mod:fptr, mod:cvar, and mod:cptr capture preconditions P3, P8, P13, and P17, respectively:

$$\begin{aligned}\text{mod:radd} &\Leftrightarrow \mathrm{modify}(retnadd)\\ \text{mod:fptr} &\Leftrightarrow \mathrm{modify}(funcptr)\\ \text{mod:cvar} &\Leftrightarrow \mathrm{modify}(ctrlvar)\\ \text{mod:cptr} &\Leftrightarrow \mathrm{modify}(ctrlptr).\end{aligned}$$

Executable buffer overflow vulnerabilities have other similarities. For example, both P4 and P9 involve jumping into either a stack or heap, and characteristics P5 and P10 involve executing instructions in the stack or heap. We use the variables $stack$ and $heap$ to reference the different types of memory, and the functions $\mathrm{jump}()$ and $\mathrm{execute}()$ to determine whether a process may jump to and execute instructions in the stack or heap. This allows us to capture preconditions P4, P9, P5, and P10 with the following characteristics:

$$\begin{aligned}\text{jmp:stack} &\Leftrightarrow \mathrm{jump}(stack)\\ \text{jmp:heap} &\Leftrightarrow \mathrm{jump}(heap)\\ \text{exe:stack} &\Leftrightarrow \mathrm{execute}(stack)\\ \text{exe:heap} &\Leftrightarrow \mathrm{execute}(heap).\end{aligned}$$

Finally, data buffer overflow vulnerabilities both require a variable that affects which execution path is taken at a future point of execution. Specifically, P14 is described as "the particular variable determines which execution path is to be taken at a future point in the execution of the process," and P18 is described as "the value pointed to by the particular pointer variable determines which execution path is to be taken at a future point in the execution of the process." We use the variables $ctrlvar$ and $ctrlptr$ as before, and the function $\mathrm{flow}()$ to determine whether a variable or pointer affects the execution path of the process. We capture both P14 and P18 with the characteristic flow:ctrl as follows:

$$\text{flow:ctrl} \Leftrightarrow \mathrm{flow}(ctrlvar).$$

Using characteristics and pseudocode to describe vulnerabilities allows us to precisely capture the similarities between different vulnerability types. For example, all buffer overflow vulnerability types contain the len:buff characteristic. Indirect buffer overflow vulnerabilities require the con:addr characteristic, and modify a pointer (mod:fptr or mod:cptr). We summarize all of these characteristics and the associated preconditions in Table 1.

We define classes of buffer overflow vulnerabilities by mapping the preconditions for each type of buffer overflow with the associated characteristic. For example, a direct executable buffer overflow has preconditions P1, P2, P3, P4, and P5. The associated set of characteristics for a direct and indirect executable buffer overflow are:

$$\begin{aligned}\text{dir:exec} = \{&\text{len:buff}, \text{con:addr}, \text{con:inst}\\ &\text{mod:radd},\ \text{jmp:stack}, \text{exe:stack}\}\\ \text{ind:exec} = \{&\text{len:buff}, \text{con:addr}, \text{mod:fptr}\\ &\text{jmp:heap}, \text{exe:heap}\}.\end{aligned}$$

We repeat this process for the remaining buffer overflow vulnerability types to obtain:

$$\begin{aligned}\text{dir:data} &= \{\text{len:buff}, \text{con:ctrl}, \text{mod:cvar}, \text{flow:ctrl}\}\\ \text{ind:data} &= \{\text{len:buff}, \text{con:addr}, \text{mod:cptr}, \text{flow:ctrl}\}.\end{aligned}$$

We use these *basic characteristic sets* as a basis for classification [14], [25]. For example, any vulnerability that contains the len:buff, con:addr, mod:cptr, and flw:ctrl characteristics will be classified as an indirect data buffer overflow. However, if any one characteristic is missing, that type of buffer overflow vulnerability is not exploitable. We discuss the necessity and sufficiency of these characteristics next.

5 Analysis

We now show that the above characteristics are both necessary and sufficient for the four types of buffer overflows to exist and be exploitable. If the basic characteristic

TABLE 1
Buffer Overflow Characteristics, Preconditions, and Vulnerabilities

Characteristic	Pseudo-Code	Preconditions	Vulnerabilities
len:buff	$\text{len}(input) > \text{len}(buffer)$	P1, P6, P11, and P15	Executable (Both), Data (Both)
con:addr	$\text{contains}(input, \text{type}(addr))$	P2, P7, and P16	Executable (Both), Data (Indirect)
con:inst	$\text{contains}(input, \text{type}(inst))$	P2	Executable (Direct)
con:ctrl	$\text{contains}(input, \text{type}(ctrlvar))$	P12	Data (Direct)
mod:radd	$\text{modify}(retnadd)$	P3	Executable (Direct)
mod:fptr	$\text{modify}(funcptr)$	P8	Executable (Indirect)
mod:cvar	$\text{modify}(ctrlvar)$	P13	Data (Direct)
mod:cptr	$\text{modify}(ctrlptr)$	P17	Data (Indirect)
jmp:stack	$\text{jump}(stack)$	P4	Executable (Direct)
jmp:heap	$\text{jump}(heap)$	P9	Executable (Indirect)
exe:stack	$\text{execute}(stack)$	P5	Executable (Direct)
exe:heap	$\text{execute}(heap)$	P10	Executable (Indirect)
flow:ctrl	$\text{flow}(ctrlvar)$	P14 and P18	Data (Both)

sets for each type of buffer overflow vulnerability are both necessary and sufficient, then disabling any single characteristic disables the associated vulnerability [14]. Furthermore, if the characteristic is removed from the system entirely, the associated class of buffer overflow vulnerabilities are also disabled. We focus on direct executable buffer overflows in our discussion, but the necessity and sufficiency of the characteristics for other types of buffer overflows may be demonstrated in a similar manner.

5.1 Necessity and Sufficiency

Consider the characteristics for a direct executable buffer overflow. We first establish necessity.

- **len:buff.** *The length the (possibly transformed) input string is longer than that of the buffer.* The necessity of this characteristic is clear: if the length of the input string is not longer than that of the buffer, the input string will fit within the buffer, and no overflow occurs.
- **con:addr and con:inst.** *The input (and possibly transformed) string may contain instructions and addresses.* Assume the contrary. The return address is the datum to be altered. It is the address of a particular instruction that is to be executed, and the succeeding instructions are then to be executed. If the return address is overwritten with data that is not an address, it cannot transfer control to the desired instruction when the function returns. This may cause the program to terminate, but that is not an executable buffer overflow attack. Hence, the input string must contain a valid address with which to overwrite the location storing the return address.
- **mod:radd.** *Input can change the stored return address without the change being countered.* Again, assume the contrary. If the stored return address cannot be overwritten, control cannot be transferred to the input instructions using this particular attack. If the change is countered, the process takes action to counter the attack (such as terminating or causing some other exception). Hence, both parts of this characteristic must hold.
- **jmp:stack.** *The program can jump to memory in the stack.* When the return address is loaded into the program counter, it will contain the address of a location on the stack (specifically, one in the buffer that was overflowed). If control cannot be transferred to that location, the instructions cannot be executed. Hence for the attack to succeed, the process must be able to transfer the flow of control to a location in the stack.
- **exe:stack.** *The program can execute instructions stored in the stack.* If the program cannot execute instructions on the stack, the input instructions will not be executed as they reside on the stack. The attack requires they be executed. This characteristic ensures they can be.

Seeing that these characteristics are sufficient for an attacker to use a direct executable buffer overflow to compromise a system merely requires one to follow the derivation of the characteristics from the preconditions of a successful attack. We note that there is an assumption that the executable instructions that are input will cause a violation of the security policy of the site. But, given that the program performs some security-related action such as temporarily granting privileges to the process (as do `setuid` and `setgid` programs in the UNIX and Linux operating systems, for example), this assumption holds.

5.2 Defenses against Buffer Overflow

Given that these characteristics must hold for the various buffer overflows to occur, the characteristics suggest a natural way to derive defenses for these attacks. We present defenses based on the individual characteristics.

Consider first the direct executable buffer overflows. A defense is sufficient to prevent the attack if it negates the characteristic. That is, if the characteristic can be established not to hold, the particular attack will fail. As before, consider each characteristic separately.

- **len:buff.** *The length of the input (and possibly transformed) string is longer than that of the buffer.* To negate this characteristic, the process must never accept any input that exceeds the buffer length. Range-checking, bounds checking, and hardware segmentation may be used to prevent the introduction of input longer than the receiving buffer. As this characteristic is common to the four types of buffer overflow discussed in this paper, negating this characteristic eliminates all four types of attacks.

- **con:addr and con:inst.** *The input (and possibly transformed) string may contain instructions and addresses.* This characteristic is more difficult to handle. The problem is in the distinction between legitimate input, and instructions and data. How can one tell if input is data, or if it is instructions and/or addresses? This is architecture and implementation dependent. On many systems instructions are binary and for many programs data is expected to be ASCII (or Unicode) characters. However, exceptions abound; for example, the location 0x61626364 is a valid location on some architectures, yet corresponds to the character string "abcd" in the ASCII character scheme. If arbitrary binary data is to be supplied, distinguishing among instructions, data, and addresses is simply infeasible. The best that can be done to negate this characteristic is to assert that when input data is to be of a particular form (such as characters making up a text message or a file name) that the input is checked *before* it is placed into the buffer. Hence, negating this characteristic requires knowledge specific to both the architecture and possibly operating system of the computer being used as well as to the specific domain of the software being scrutinized.
- **mod:radd.** *Input can change the stored return address without the change being countered.* Several approaches can negate this characteristic. The first is to store the return address in a memory location protected from being altered by the running process. This requires either a special architecture with tag bits, or a special memory page or segment with write permission turned off. A second approach is to store the return address in a location other than the program stack. This approach requires that the "Return Address Stack (RAS)" (for want of a better name) be in a different segment than all program variables, so any attempt to overflow into it will cause a fault. A third approach is to store a copy of the return address in a variable as part of the prologue of the function call, and then when the function is to return, have its epilogue compare the return address on the stack with the stored return address; if the two differ, the return address has been altered. Again, note these checks and changes can be implemented in either hardware or software.
- **jmp:stack.** *The program can jump to memory in the stack.* The negation of this characteristic is similar to classical address bounds checking. The system notes the addresses of memory that belong to the program stack. Before the return address is placed into the program counter, it is checked to determine whether it lies within the memory allocated to the stack. If so, a fault occurs. Again, the checking can be implemented in either hardware or software.
- **exe:stack.** *The program can execute instructions stored in the stack.* The negation of this characteristic is similar to a technique used in older architectures. In those systems, instructions (called "text") were stored in an executable portion of memory, and data (including the stack) was stored in a nonexecutable portion of memory. Modern architectures can use execution privileges on segments and pages to disable the ability to execute data as instructions.

Continuing with other types of buffer overflows, we note that many of the characteristics are the same as for the direct executable buffer overflow. We present ameliorations only for those that differ.

- **mod:fptr.** *Input can change the value in the function pointer variable without being detected (indirect executable buffer overflow).* This is similar to the return address being changed without detection (mod:radd). However, there are two key differences. The first is that the function pointer is a variable, not an element of the program state, so the computer system cannot store it in a special area without being told to do so. To the underlying architecture, the function pointer is simply a program variable. In order to store the function pointer into a different segment of memory, the compiler must generate special directives. Hence, applying the countermeasure of storing the function pointer in a special area of memory requires that the compiler be modified to indicate this.

  The second difference is the ability to store the pointer in read-only memory. If that were done, only the operating system could modify it. This is not feasible simply because the function pointer is program data, and the process will modify the value during the execution of the program—otherwise, it should be a constant and can be stored in read-only memory. An interesting approximation to this scheme is to notice that changes to the value of the function pointer should occur only by assignment to that memory location (either by directly using the variable or indirectly using a pointer). The compiler could "wrap" each such assignment with code that turns off read-only permission to that location or page, performs the assignment, and then restores read-only permission. This way, should a buffer overflow attempt to overwrite that location, the read-only access would detect the attempted write and object.

  The fact that the program controls the variables suggests an approach based on Biba's integrity model [26]. Define two integrity classes *Untrusted* and *Trusted*. Any variable to which the program assigns a value is placed into the *Trusted* class. Any variable with a value that depends on input (whether the input be from a user, the environment, or some other source not under the program's control) is placed into the *Untrusted* class. The *Trusted* class dominates the *Untrusted* class. Whenever a program uses a value stored in a variable, it checks the class of that variable. If the variable is *Trusted*, the value is used. If the variable is *Untrusted*, the process must check the value to ensure it is acceptable, and change the class of the variable to *Trusted*, before it is used. If the check fails, or the process cannot check the value, the program stops.

Applying this idea to the buffer overflow problem, initially the function pointer value is *Trusted*, because it is set by the process. The input data values are not under the control of the program. So, each element added to (and beyond) the buffer is *Untrusted*. When the value in the function pointer variable is overwritten, its class is also overwritten and marked *Untrusted*. Then, when the program references the value in the function pointer, it notes that the value is *Untrusted* and there is no check procedure to change the class to *Trusted*. Hence, the program aborts.

- **jmp:heap.** *The program can jump to memory in the heap (indirect executable buffer overflow).* The techniques to prevent this characteristic from holding are the same as for preventing jumps to memory in the stack (jmp:stack), using the addresses of memory making up the heap.
- **exe:heap.** *The program can execute instructions in the heap (indirect executable buffer overflow).* The techniques for this characteristic are the same as those for preventing execution of instructions stored on the stack (exe:stack).
- **con:ctrl.** *The input (and possibly transformed) string may contain data of the type of the particular variable (direct data buffer overflow).* This characteristic imposes a constraint more onerous than distinguishing among data, instructions, and addresses, although the fundamental underlying problem is the same. How does one distinguish between the integer byte 0x61 and the character "a" on a Linux or UNIX system? The problem is that if the bit pattern is interpreted as an integer, it represents the decimal number 97; if it is interpreted as a character, it represents the letter "a." As with distinguishing among data, instructions, and addresses, if the expected type can be characterized in a form that can be checked, then the amelioration is to perform the checking before placing the data into the buffer. This problem is discussed above, in con:addr and con:inst. The integrity-based technique discussed in mod:fptr also works here.
- **mod:cvar and mod:cptr.** *The value stored in the particular variable can be changed without being detected (direct data buffer overflow).* The techniques for this characteristic are the same as for those of detecting changes in the value of function pointer variables (mod:fptr). Both the particular variable and the function pointer variable are variables in the process space, so can be treated identically.
- **flow:ctrl.** *The particular variable determines which execution path is to be taken at a future point in the execution of the process (direct data buffer overflow).* This characteristic is a function of the program being exploited. This situation can be detected only when checks are placed upon the program so that the "correct path of control" is taken. The problem, of course, is determining the "correct path of control." If the setting of a variable determines which of two paths may be taken, there are circumstances in which the selection of either path may be correct. Determining which is correct given the expected state of the program, and then comparing that to the path actually taken, is an interesting process-level problem in anomaly based intrusion detection.

  Similarly, an integrity-based technique such as the one discussed under mod:fptr would apply for this characteristic when the variable controlling the path to be taken is to be set by the program and not by user input. However, if the variable is to be set by user input (for example, a string that the user enters or that is taken from the environment), then the process cannot determine whether the *Untrusted* data is what was originally set or entered. Predicating the flow of control in a security-related program, which should have high integrity and assurance, upon untrusted data is at best careless programming and in general, is an invitation to compromise.

6 Examples

Researchers have presented many methods of combating the buffer overflow problem. This section casts the goals of those methods into terms of our characteristics, examines how precise they are, and suggests alternate methods for handling the buffer overflow problem.

6.1 Segmentation

The memory management technique of segmentation, in which functions and variables are assigned to different segments of memory, offers a simple way to negate characteristic len:buff. If each buffer is placed into its own segment, any attempt to overflow the buffer will cause a segment fault, resulting in a trap.

6.2 Integer Analysis to Determine Buffer Overflow

Wagner et al. [27] developed a technique that infers constraints on the ranges of variable values. This technique treats strings as an abstract data type and models buffers as pairs of integers, namely the allocated size and the number of bytes currently in the buffer. It then traverses the program's parse tree and develops a system of integer range constraints. Once the ranges for all variables have been inferred, the technique checks a safety property for each string. If the analysis results in the string's length lying in the range $[a, b]$ and the buffer's allocated size lying in the range $[c, d]$, then (assuming a downward-growing stack):

1. if $b \leq c$ then the string never overflows the buffer;
2. if $a > d$ then a buffer overflow will always occur upon any execution involving that string; and
3. if the ranges overlap then a buffer overflow may occur.

The intent of this technique is to detect characteristic len:buff holding, and if so report the problem. Thus, it deals with all types of buffer overflows. As the technique is static, both false positives and false negatives are possible. The authors point out that the tool is a prototype, and so suffers from several limits (such as not handling many pointer issues, for example pointer aliasing), and its implementation could be improved.

6.3 STOBO

Haugh and Bishop [28] extended the integer analysis model to include dynamic analysis. In this approach, the integer constraints are developed and embedded into the program. They are not analyzed until runtime. Whenever an operation involving a buffer or string occurs, the appropriate constraints are checked using runtime data. This allows checking of (most kinds of) pointers, and allows reporting of potential buffer overflows even if the particular input data does not cause an overflow to occur.

Like the integer analysis approach, this approach speaks to characteristic len:buff. It deals with all types of buffer overflows. Because the method is dynamic, it gives fewer false positives and negatives than does the static analysis technique.

6.4 Type-Assisted Buffer Overflow Detection

Lhee and Chapin [29] developed a technique to perform range checking on buffers that the program references. This is done at runtime, and requires adding a data structure to the GNU C compiler that describes the type of automatic and static buffers the types of which are known at compile time. For dynamically allocated (heap) objects, the method uses a table that tracks the heap objects and their sizes. The process then uses these data structures to perform range checking of arguments to vulnerable string functions in the C library.

As with the integer analysis techniques, this method tries to negate characteristic len:buff. Although it focuses on all types of buffer overflows, the particular method is based upon the assumption that overflows arise with string manipulation functions. This means that overflows arising from non-ASCII data will not be caught. Nevertheless, in the environment which the work was done (that of C programming), the assumption is valid enough so that the authors were able to block a large class of buffer overflow attacks.

6.5 CRED

The C Range Error Detector [30] implements a method of checking for out-of-bounds addresses that leads to the detection of buffer overflow attacks. The program replaces out-of-bounds addresses stored in pointers with the address of an object in the heap (called the "out-of-bounds object"). When a pointer is dereferenced, it is checked to see if it is out-of-bounds. If so, the program terminates with an error message.

As with the previous three techniques, this method detects characteristic len:buff holding. Hence, it is suitable for all buffer overflows. Although the authors of the tool took pains to ensure that out-of-bounds addresses could be used in arithmetic operations and comparisons, they did not address type punning. If a pointer were cast to an integer, then used as an operand in an arithmetic operation, and then recast to a pointer, an error may arise if the new pointer is dereferenced because it could point to a legitimate object. Similarly, out-of-bounds pointers may be passed to library functions.

6.6 Jump Pointer Control

Hardware/Software Address Protection (HSAP) [31] focuses on memory locations that store an address to a code segment: return addresses and function pointers. HSAP handles them differently.

In the case of return addresses, HSAP adds hardware bounds checking. When a return address is popped, before it is put in the program counter, the system checks that the address is equal to or greater than the value of the frame pointer. If so, the return address is returning into the stack, enabling characteristic jmp:stack. The system aborts the process.

HSAP uses a special hardware register containing a random key to handle function pointers. A key is randomly assigned to this special register. Every function pointer value is XORed with this key when stored. When a function pointer is invoked, a special jump instruction XORs the value in the variable with the value in the register. If the value in the function pointer was assigned by the process, the resulting address is that of the right function. If not, the address will be invalid and cause the program to terminate. This ensures characteristic mod:fptr will be false.

HSAP handles both types of executable buffer overflows. It does not handle data buffer overflows. However, if the process were modified to use the random register for all pointer variables, HSAP would also block the indirect data buffer overflow (specifically, characteristic mod:cptr). It is not clear if this technique could be generalized to handle direct data buffer overflows. The problem is how to make the assignments to variables when input occurs. The data that is written out of bounds would have to not use the special register, and the data written in bounds would have to use that register. But if this distinction could be made, the use of the register would be unnecessary.

6.7 StackGuard, MemGuard, and PointGuard

StackGuard [32] is designed to thwart direct executable buffer overflows. It inserts a *canary*, or random number computed at runtime, into the stack between the return address and any variables. Before the function returns, the canary is popped and compared with its original value. If the two differ, the canary has been altered, and (presumably) the return address has also been altered. The process is then terminated.

StackGuard works because it detects a violation of characteristic mod:radd. It asserts that the return address has been changed, and therefore a buffer overflow occurs. If a buffer overflow attack were used to alter the return address, the attacker would need to overlay the canary with data containing the value of the original canary. As the canary is generated randomly, this is highly unlikely unless the attacker can observe the executing process (and if the attacker can do so, the attacker probably does not need to compromise the system using a buffer overflow attack).

Although the design and intent of StackGuard is to check for changes to the return address stored on the stack, StackGuard does not actually do so. It instead checks a word near the return address. This allows both false positives and false negatives to occur. A false positive may occur if, for example, the data used in the buffer overflow is long enough to overwrite the canary but not long enough to overwrite the return address. A false negative may occur if the canary's value is not altered, as mentioned above. An alternate approach addresses these problems. Rather than generating a canary, determine the value of the return address before the function is invoked, and store that in memory (not on the stack) before the call. Then, check the value of the return address itself before executing the return. This eliminates false negatives, because if the return address has changed, characteristic mod:radd is satisfied, so every executable buffer overflow will be detected. It does not eliminate false positives, because the return address may change for reasons other than buffer overflow attacks (perhaps the program does so as part of an unusual but planned change of the flow of control). But it reduces the number of possible false positives compared to StackGuard, and could be implemented by changing the function prologue and epilogue routines in the GNU C compiler.

MemGuard, described in the same paper, is a generalization of StackGuard that uses virtual memory protection mechanisms to protect specific memory locations such as the return address. MemGuard allows a precise implementation of characteristics mod:radd, mod:fptr, mod:cvar, and mod:cptr, because the specific location of the return address (or variable) can be marked as immutable, so any attempt to change it will cause a trap. However, experiments showed the overhead of MemGuard is unacceptably high.

PointGuard [33] is a generalization of StackGuard. PointGuard places canaries next to all function pointers, and whenever a function pointer is dereferenced, the canary is validated. The techniques used are otherwise the same as for StackGuard. PointGuard attempts to detect characteristics mod:fptr and mod:cptr, and suffers from the same problems as StackGuard.

6.8 Split Control and Data Stacks

SmashGuard [33] provides microarchitectural hardware support to detect changes to the return address. It copies the return address and frame pointer to a small hardware stack as well as the process stack. On return, the return address from the process stack is compared to that on the hardware stack. If they differ, the program terminates.

This method handles direct executable buffer overflows only. It negates characteristic mod:radd by detecting the change to the return address. The discussion of countermeasures explains why this approach does not generalize well to the other types of buffer overflow attacks. Further, the implementation of SmashGuard must keep the additional hardware stack in an area that the process cannot alter.

An alternate implementation is to do this in software rather than hardware [34]. The software implementation stores a copy of the return address on a stack specially allocated by the compiler. The prologue of a function call is altered to save the return address; the epilogue is altered to compare the two values. The hardware version, designed for greater efficiency, causes the placement of the return address on the second stack, and its comparison with the other stored return address, to occur as part of the call and return operations. Both methods cause program termination if the comparison fails. This checks for characteristic mod:radd, and negates it when it is found to hold.

Minezone RAD and read-only RAD [35] use variants of this technique. Minezone RAD allocates “guard pages” that are write-protected before and after the memory allocated to the special stack to hold return addresses. Read-only RAD sets the location holding the pushed address to read-only after it is pushed onto the stack. Gadaleta et al. [36] examine changes to the call and return instructions that have the same effect as read-only RAD. All these techniques split the control and data stacks, and so are categorized as negating characteristic mod:radd.

6.9 Secure Return Address Stack (SRAS)

Branch prediction applied to function return enables an interesting technique to detect buffer overflow [34]. This method is based on the Return Address Stack used in modern processors. In normal operations RAS mispredictions are the result of speculative updates of the stack and overflows due to limited RAS size. But buffer overflows can cause these mispredictions. The difference between the two is that when a buffer overflow is the cause, an exception handler cannot trace back to the previous stack frame, the address of which will have been overwritten as a side effect of the overflow. Such a handler incurs high overhead, and the authors develop a way to eliminate the speculative nature of the predictions, obviating the need for the indirect referencing of the exception handler. This attempts to negate characteristic mod:radd by detecting changes to the return address.

6.10 Other Defenses

Address Space Layout Randomization (ASLR) negates characteristics, mod:radd, mod:fptr, mod:cvar, and cptr by randomizing the locations at which those variables are stored [37], [38], [39], [40]. ASLR also has been applied to rearrange code, which negates jmp:stack and jmp:heap. The rearranging effectively prevents the attacker from knowing what address needs to be changed, and in many cases the address would no longer be accessible using a buffer overflow.

A similar idea is to randomize instruction sets on a per-process basis [41]. For example, every instruction is *xor*ed with an encoding key, When the instruction is to be executed, it is first *xor*ed with the corresponding decoding key. Should an attacker attempt a direct executable buffer overflow, it will fail because the injected code is not *xor*ed with an encoding key. This negates the characteristics exe:stack and exe:heap.

Heap spraying [18] replicates the attack code in numerous places in memory. NOZZLE [42] statically analyzes objects on the heap to see if jumping into them cause shell code to execute. Egele et al. [43] use instruction emulation to identify the buffers that contain shellcode. In terms of characteristics, both detect that the characteristic con:inst holds, and take appropriate action. BuBBle [44] inserts special instructions in string memory. When the string is read, the mechanism reconstructs the original string. If execution is attempted, the special instructions cause an exception. These methods effectively prevents the execution of input code on the heap, negating the characteristic exec:heap.

A Structured Exception Handling (SEH) overwrite [45] overwrites a (function) pointer to an exception handler and then triggers the exception. Microsoft’s defense, called a Structured Exception Handling Overwrite Protection (SEHOP), checks that the exception handler list is intact before allowing an exception to be invoked [46]. Both SEH and SEHOP speak to the characteristic mod:fptr.

Several versions of the UNIX and Linux systems disable execute permission for the stack [47], [48]. This counters characteristic exe:stack, by preventing the program from executing instructions on the stack.

6.11 Summary

The focus of the techniques surveyed lies upon four sets of characteristics. The first set is that the length of the input is longer than that of the buffer (characteristic len:buff). The second set is that a value—the return address or a function pointer—is altered (characteristics mod:radd, mod:fptr, and mod:cptr). The third negates exe:stack, the ability to execute instructions on the stack. The fourth detects branches into the stack (characteristic jmp:stack). That leaves several other preconditions unexplored.

The difficulties in ensuring that characteristics con:addr, con:inst, and con:ctrl do not hold are discussed above. Strict typing that differentiates between data, addresses, and instructions is a step toward negating these preconditions. If *all* input is of type data, then the execution buffer overflows and the indirect data buffer overflows are eliminated because characteristic con:addr is false.

One technique (HSAP) checks for return addresses within the stack, but not for function pointers pointing to the heap or to data areas. If instructions were loaded into a data area, in most systems they could not be executed as the data area has execute permission turned off. But many dynamic loading systems load functions into the heap when they are invoked. On systems like these, turning off execute permission for the memory making up the heap would inhibit dynamic loading. So, it is legitimate for a function pointer to point into the heap if dynamic loading is used. Various heuristics could help determine if this were true for any particular process, but the complexity of the analysis is probably greater than methods that negate other preconditions. Hence, negating characteristic jmp:heap is typically not done.

Characteristic mod:cvar asks whether a value has been modified to cause an action that violates the security policy. This requires certain simplifying assumptions. The simplest assumption is that the data is only to be modified by the program; this leads to the approach based on Biba's model discussed earlier.

Characteristic flow:ctrl is a function of the program, and say that if a buffer overflow changes a variable that does not affect the flow of control, it has no security implications *for that program*. Note that if the value is output and a second program acts based on that output, the value is affecting the flow of control of the composition of the programs, instantiating characteristic flow:ctrl. As all programs involving security do rely on variables, it is infeasible to negate this characteristic taken alone.

7 Conclusion and Future Work

This paper provides a methodology for analyzing buffer overflow vulnerabilities. It derives specific *preconditions* that must hold in order for an attacker to exploit a buffer overflow vulnerability, and then reframes them in terms of well-defined *characteristics*. These are both necessary and sufficient for the buffer overflows to be exploitable.

The preconditions distinguish among four distinct types of buffer overflows: direct executable, indirect executable, direct data, and indirect data. These have common characteristics as well as different characteristics. This suggests two approaches.

The first approach is to examine the sets of characteristics, and focus on those common to all types of buffer overflows. The first is that the length of the input be not greater than the length of the buffer. So range-checking compilers, and other tools that check for out-of-bounds references, work against all buffer overflow attacks. The second is that the data being loaded must be treated as two different types: data (for the input) and addresses or instructions (for the use). This precondition is harder to detect for the class of direct data buffer overflows, because the defense must distinguish between the type of the input data and the type of the data in the variable. An approach using Biba's integrity model was briefly discussed.

The other characteristics vary among the different types of buffer overflows. Approaches to implementing detection and prevention methods for each were discussed, and a number of tools that detect buffer overflow attacks were classified based on the characteristics they negated.

Our characteristics focus solely on the technical aspects of the programs. As noted, it assumes that a buffer overflow will cause a security breach. This is not necessarily true. If the process is running with the attacker's privileges, an executable buffer overflow attack will gain the attacker nothing. Incorporating policy and other environmental elements into the characteristics in a way that an analyst can test would greatly enhance this work.

The second approach is to tie the characteristics into attack models such as the requires/provides attack model [49], and thus link vulnerabilities with attack models firmly. In the event that a site encode parts of its policy into a language capturing this type of model, an automated reasoning engine could determine what vulnerabilities were consistent with the site's policy.

Finally, the development of automated tools to examine systems for the presence of characteristics would provide an effective way of detecting potential security vulnerabilities. Preconditions underlie characteristics, which in turn underlie vulnerabilities, and many are common to more than one class of vulnerability. This would effectively allow us to check for vulnerabilities by building on the results found during checking for other vulnerabilities, making the process considerably more effective.

Acknowledgments

The authors thank Duilio Protti, Marco Ramilli, and Mevlüt Türker Garip for their valuable comments. This work was supported in part by grant CCF-0905503 from the US National Science Foundation (NSF) to the University of California at Davis. Any opinions expressed are not necessarily those of the NSF.

References

[1] M.W. Eichin and J.A. Rochlis, "With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988," *Proc. IEEE Symp. Security and Privacy*, pp. 326-343, 1989.
[2] D. Seeley, "A Tour of the Worm," *Proc. Winter USENIX Conf.*, pp. 287-304, 1989.
[3] E.H. Spafford, "Crisis and Aftermath," *Comm. ACM*, vol. 32, no. 6, pp. 678-687, 1989.
[4] CERT, "W32/Blaster Worm," CERT Advisory CA-2003-20, http://www.cert.org/advisories/CA-2003-20.html, Aug. 2003.

[5] CERT, "MS-SQL Server Worm," CERT Advisory CA-2003-04, http://www.cert.org/advisories/CA-2003-04.html, Jan. 2003.

[6] CERT, "Apache/mod_ssl Worm," CERT Advisory CA-2002-27, http://www.cert.org/advisories/CA-2002-27.html, Oct. 2002.

[7] CERT, "'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL," CERT Advisory CA-2001-19, http://www.cert.org/advisories/CA-2001-19.html, July 2001.

[8] CERT, "Continuing Threat of the 'Code Red' Worm," CERT Advisory CA-2001-23, http://www.cert.org/advisories/CA-2001-23.html, July 2001.

[9] S. Christey, "Common Vulnerabilities and Exposures," http://cve.mitre.org, Apr. 2011.

[10] R. Abbott, J. Chin, J. Donnelley, W. Konigsford, S. Tokubo, and D. Webb, *Security Analysis and Enhancements of Computer Operating Systems,* ICET, Nat'l Bureau of Standards NBSIR 76-1041, Apr. 1976.

[11] T. Aslam, "A Taxonomy of Security Faults in the Unix Operating System," master's thesis, Dept. of Computer Sciences, Purdue Univ., Aug. 1995.

[12] R. Bisbey II and D. Hollingsworth, "Protection Analysis: Final Report," Technical Report ISI/SR-78-13, Information Sciences Inst., Univ. of Southern California, May 1978.

[13] C.E. Landwehr, A.R. Bull, J.P. McDermott, and W.S. Choi, "A Taxonomy of Computer Program Security Flaws," *ACM Computing Surveys,* vol. 26, no. 3, pp. 211-254, 1994.

[14] M. Bishop, "Vulnerability Analysis," *Proc. Second Int'l Symp. Recent Advances in Intrusion Detection,* pp. 125-136, 1999.

[15] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," *Proc. Foundations of Intrusion Tolerant Systems,* pp. 227-237, 2003.

[16] AlephOne, "Smashing the Stack for Fun and Profit," *Phrack,* vol. 7, no. 49, 1996.

[17] M. Conover, "w00w00 on Heap Overflows," http://www.cgsecurity.org/exploit/heaptut.txt, 1999.

[18] Skypher, "Internet Explorer IFRAME src&name Parameter BoF Remote Compromise," http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/bjwever/advisory_iframe.html.php, 2004.

[19] D. Blazakis, "Interpreter Exploitation: Pointer Interference and JIT Spraying," technical report, Semantiscope, http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf, 2010.

[20] A. Sotirov, "Heap Feng Shui in Javascript," *Proc. Black Hat Europe,* http://www.blackhat.com/presentations/bh-europe-07/FSotirov/Presentation/bh-eu-07-sotirov-apr19.pdf, 2007.

[21] S. Esser, "Samba 3.x QFILEPATHINFO Unicode Filename Buffer Overflow," e-matters GmbH, Security Advisory 13/2004, Nov. 2004.

[22] T. Newsham, "Format String Attacks," technical report, Guardent, Inc., Sept. 2000.

[23] *AMD64 Architecture Programmer's Manual Volume 2: System Programming,* Advanced Micro Devices, June 2010.

[24] *Intel®64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1,* Intel Corp., May 2007.

[25] S. Engle, "A Policy-Based Vulnerability Analysis Framework," PhD dissertation, Dept. of Computer Science, Univ. of California, Davis, June 2010.

[26] K.J. Biba, "Integrity Considerations for Secure Computer Systems," Technical Report MTR-3153, The MITRE Corporation, June 1975.

[27] D. Wagner, J.S. Foster, E. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. Symp. Network and Distributed Systems Security,* pp. 3-17, 2000.

[28] E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," *Proc. Network and Distributed System Security Symp.,* pp. 123-130, 2003.

[29] K.-S. Lhee and S.J. Chapin, "Buffer Overflow and Format String Overflow Vulnerabilities," *Software: Practice and Experience,* vol. 33, no. 5, pp. 423-460, 2003.

[30] O. Ruwase and M.S. Lam, "A Practical Dynamic Buffer Overflow Detector," *Proc. Symp. Network and Distributed System Security,* pp. 159-169, 2004.

[31] Z. Shao, Q. Zhuge, Y. He, and E. Sha, "Defending Embedded Systems against Buffer Overflow via Hardware/Software," *Proc. 19th Ann. Computer Security Applications Conf.,* pp. 351-361, 2003.

[32] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proc. Seven USENIX Security Symp.,* 1998.

[33] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard^{TM}: Protecting Pointers from Buffer Overflow Vulnerabilities," *Proc. 12th Conf. USENIX Security Symp.,* pp. 91-104, 2003.

[34] J. Xu, Z. Kalbarczyk, S. Patel, and R.K. Iyer, "Architecture Support for Defending against Buffer Overflow Attacks," *Proc. Workshop Evaluating and Architecting System Dependability,* 2002.

[35] T.-C. Chiueh and F.-H. Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks," *Proc. 21st Int'l Conf. Distributed Computing Systems,* pp. 409-417, 2001.

[36] F. Gadaleta, Y. Younan, B. Jacobs, W. Joosen, E.D. Neve, and N. Beosier, "Instruction-Level Countermeasures against Stack-Based Buffer Overflow Attacks," *Proc. First EuroSys Workshop Virtualization Technology for Dependable Systems,* pp. 7-12, 2009.

[37] S. Bhatkar, R. Sekar, and D.C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," *Proc. 14th USENIX Security Conf.,* pp. 255-270, 2005.

[38] S. Chen, J. Xu, Z. Kalbarczyk, and R.K. Iyer, "Security Vulnerabilities: From Analysis to Detection and Masking Techniques," *Proc. IEEE,* vol. 94, no. 2, pp. 407-418, Feb. 2006.

[39] S. Forrest, A. Somayaji, and D.H. Ackley, "Building Diverse Computer Systems," *Proc. Sixth Workshop Hot Topics in Operating Systems,* pp. 67-72, 1997.

[40] J. Xu, Z. Kalbarczyk, and R.K. Iyer, "Transparent Runtime Randomization for Security," *Proc. 22nd Int'l Symp. Reliable Distributed Systems,* pp. 260-269, 2003.

[41] G.S. Kc, A.D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomiztion," *Proc. 10th ACM Conf. Computer and Comm. Security,* pp. 272-280, 2003.

[42] P. Ratanaworabhan, B. Livshits, and B. Zorn, "NOZZLE: A Defense against Heap-Spraying Code Injection Attacks," *Proc. 18th USENIX Security Symp.,* pp. 169-186, 2009.

[43] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending Browsers against Drive-by Downloads: Mitigating Heap-Spraying Code Injection Attacks," *Proc. Sixth Int'l Conf. Detection of Intrusions and Malware, and Vulnerability Assessment,* pp. 88-106, 2009.

[44] F. Gadaleta, Y. Younan, and W. Joosen, "BuBBle: A Javascript Engine Level Countermeasure against Heap-spraying Attacks," *Proc. Second Int'l Symp. Eng. Secure Software and Systems,* pp. 1-17, 2010.

[45] D. Litchfield, "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server," White Paper, NGS Secure http://www.nccgroup.com/Libraries/Document_Downloads/Defeating_the_Stack_Based_Buffer_Overflow_Prevention_Mechanism_of_Microsoft_Windows_2003_Server.sflb.ashx, Sept. 2003.

[46] M. Miller, "Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP," http://www.blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx, 2009.

[47] S. Designer, "Linux Kernel Patch from the Openwall Project," http://www.openwall.com/linux/README.shtml.

[48] C.H.S. Dik, "RE: Binary Security Attacks," news:comp.security.unix, 2011.

[49] S.J. Templeton and K. Levitt, "A Requires/Provides Model for Computer Attacks," *Proc. New Security Paradigms Workshop,* pp. 31-38, 2000.

Matt Bishop received the PhD degree in computer science from Purdue University, where he specialized in computer security, in 1984. His main research area include the analysis of vulnerabilities in computer systems. His textbook, *Computer Security: Art and Science*, was published in 2002 by Addison-Wesley Professional. He is a member of the IEEE.

Sophie Engle received the PhD degree in computer science from the University of California, Davis, in 2010. Her primary research focus is on computer security, including topics such as vulnerability analysis, insider threat, and electronic voting. Her research interests also include topics such as information visualization, network and graph theory, and computer science education.

Damien Howard received the MS degree in electrical and computer engineering from the University of California, Davis, in 2007. His research focused on computer security. More specifically, his thesis concerned vulnerability analysis and attack-tool classifications. He is a member of the IEEE.

Sean Whalen received the PhD degree in computer science from the University of California, Davis, in 2010, and is currently an I3P Fellow. He applies physics-based modeling techniques to problems in computer security. His research interests include computer security, statistical inference, machine learning, and information visualization.