A Taxonomy of Buffer Overflow Preconditions

Matt Bishop, Damien Howard, Sophie Engle, and Sean Whalen

Department of Computer Science
University of California, Davis
One Shields Avenue, Davis, CA, 95616

{mabishop, djhoward, sjengle, shwhalen}@ucdavis.edu

January 2010*

Abstract

Recent work on vulnerabilities has focused on buffer overflows, in which data exceeding the bounds of an array is loaded into the array. The loading continues past the end of the array, causing variables and state information 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 non-secure state. This work develops a taxonomy of buffer overflow vulnerabilities based upon preconditions, or conditions 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.

Keywords: Protection Mechanisms, Software/Program Verification, Security and Privacy, Arrays
Index Terms: Vulnerabilities, Buffer Overflows, Protection Mechanisms, Security

1 Introduction

Buffer overflows occur when a sequence of bytes of length n is placed into a 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.

That buffer overflows can cause security problems is well documented. 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 [9, 10, 11, 12, 13, 14]. The problem continues to exist despite efforts to eliminate it.

Numerous taxonomies of vulnerabilities [15, 16, 17, 18] present classifications based on general categories such as "unexpected change" or "logic error." This work expands upon Bishop's classification proposed for buffer overflow [19]. Cowan et al. [20] analyzed techniques for defending against buffer overflows. We defer discussion of his classification scheme so we can demonstrate how 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 will be analyzed in the context of our classification scheme.

*This is an updated version of this paper, originally written in July 2006.

In this paper, we examine the causes for buffer overflow vulnerabilities existing 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 leads to ways to make potential solutions more effective in the sense of handling multiple classes of buffer overflows.

The next section of the paper provides some background that will illuminate key characteristics of the buffer overflow problem, as well as the principles underlying our classification scheme. The third section presents our classification of buffer overflows, and the fourth section examines proposed solutions in light of that classification. We conclude with some thoughts on classification criteria in general, and 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 [21] and Conover et al. [22] present detailed descriptions of how these attacks work.

2.1 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 is intended to print the number "139" into `buf`:

```
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, 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 uploads 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 [23]).

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 [24]. Hence, we distinguish between format string vulnerabilities in general and buffer overflow vulnerabilities. Our work does not address preconditions for format string overflows.

2.2 Architectural Considerations

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

The data area of process memory provides space for non-transient 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.

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 function 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 1000 on the first call to *plugh*, and an address of 12345 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 dynamic load 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 may be modified only on the stack. Changing return addresses and function pointers alter flow of control, while changing variables results in changes to data. This suggests two broad classes of buffer overflow attacks.

2.3 Data Buffer Overflow

A data buffer overflow occurs when input overwrites existing data, causing the program to act in a manner that violates security. 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 *login* program buffer overflow in UNIX Version 6. 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 uploaded data fall into this class.

2.4 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 most common 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 PC, the uploaded machine instructions execute.

A good example of this is the *fingerd* flaw that the Internet Worm of 1988 exploited [1, 18, 3]. 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.

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.

2.5 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. Direct executable buffer overflows cause the flow of control to jump to a location other than that which the program should have jumped to. Indirect buffer overflows illicitly change a pointer, causing the program to use incorrect data (in which case it is essentially a data buffer overflow with the data being present in the process memory already) or to execute instructions that should not be executed at that point in time (in which case it is essentially an execution buffer overflow with the instructions being present in the process memory already).

3 Analysis

Our goal is to establish the *preconditions* necessary for a vulnerability. To do this, we need to examine how an attacker might exploit a 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.

3.1 Executable Buffer Overflow

First, consider a buffer overflow on the stack. Here, the attacker uploads 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, causing the forbidden program to be executed.

Said succinctly, this attack is: "Upload 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 can be split into five parts:

1. Upload an input string that either contains or, after being transformed, contains:
2. 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, and
5. Executes the machine language program stored there.

This leads to the following set of preconditions, corresponding to the above parts:

P1. The length of the (possibly transformed) uploaded string is longer than that of the buffer.
P2. The uploaded (and possibly transformed) string may contain 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 upload data that contains valid instructions, but the process may not execute them. For example, some systems disable execution of portions of memory, and some architectures separate executable instructions and data.

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. Upload an input string that either contains or, after being transformed, contains:
2. 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 uploaded (and possibly transformed) string may contain 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 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. Upload an input string that either contains or, after being transformed, contains:
2. Data, including 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:

P11. The length of the (possibly transformed) string is longer than that of the buffer.

P12. The uploaded (and possibly transformed) string may contain data of the type of the particular variable.

P13. The value stored in the particular variable can be changed without being countered.

P14. 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 uploaded instructions. The instructions are not uploaded, of course, but must still be executed. There is no question that the instructions *can* be executed because the program has them in one flow of control 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 uploaded (and possibly transformed) string may contain 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.

3.3 Necessity and Sufficiency

We now show that the above preconditions are both necessary and sufficient for the above types of buffer overflows to exist and be exploitable.

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

- **The length of the (possibly transformed) uploaded string is longer than that of the buffer.** The necessity of this precondition is clear: if the length of the uploaded string is not longer than that of the buffer, the uploaded string will fit within the buffer, and no overflow occurs.
- **The uploaded (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 uploaded string must contain a valid address with which to overwrite the location storing the return address.
- **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 uploaded 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 precondition must hold.
- **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.
- **The program can execute instructions stored in the stack.** If the program cannot execute instructions on the stack, the uploaded instructions will not be executed as they reside on the stack. The attack requires they be executed. This precondition ensures they can be.

Seeing that these preconditions 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 preconditions from the (successful) attack. We note that there is an assumption that the executable instructions that are uploaded 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.

3.4 Defenses Against Buffer Overflow

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

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

- **The length of the uploaded (and possibly transformed) string is longer than that of the buffer.** To negate this precondition, 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 precondition is common to the four types of buffer overflow discussed in this paper, negating this condition eliminates all four types of attacks.

- **The uploaded (and possibly transformed) string may contain instructions and addresses.** This precondition 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 precondition 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 precondition 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.

- **Input can change the stored return address without the change being countered.** Several approaches can negate this precondition. 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" (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.

- **The program can jump to memory in the stack.** The negation of this precondition 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.

- **The program can execute instructions stored in the stack.** The negation of this condition is similar to a technique used in older architectures, such as that of the PDP-11. In those systems, instructions (called "text") were stored in an executable portion of memory, and data (including the stack) was stored in a non-executable 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 preconditions are the same as for the direct executable buffer overflow. We present ameliorations only for those that differ.

- **The uploaded (and possibly transformed) string may contain addresses.** *(Indirect Executable Buffer Overflow)* This is a sub-case of the precondition where the uploaded string may contain instructions and addresses.

- **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. However, there are two key differences. The first is 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 [25]. Define two integrity classes. Anything that the program assigns is placed into the *Trusted* class. Anything that is assigned to from 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.

- **The program can jump to memory in the heap.** *(Indirect Executable Buffer Overflow)* The techniques to prevent this precondition from holding are the same as for preventing jumps to memory in the stack, using the addresses of memory making up the heap.
- **The program can execute instructions in the heap.** *(Indirect Executable Buffer Overflow)* The techniques for this precondition are the same as those for preventing execution of instructions stored on the stack.
- **The uploaded (and possibly transformed) string may contain data of the type of the particular variable.** *(Direct Data Buffer Overflow)* This precondition 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 "the uploaded string may contain instructions and addresses." The integrity-based technique discussed in "input can change the value in the function pointer variable without being detected" also works here.

- **The value stored in the particular variable can be changed without being detected.** *(Direct Data Buffer Overflow)* The techniques for this precondition are the same as for those of detecting changes in the value of function pointer variables. Both the particular variable and the function pointer variable are variables in the process space, so can be treated identically.
- **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 precondition 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 "input can change the value in the function pointer variable without being detected" would apply for this precondition 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.

4 Examples

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

4.1 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 preconditions P1, P6, P11, and P15. If each buffer is placed into its own segment, any attempt to overflow the buffer will cause a segment fault, resulting in a trap.

4.2 Integer Analysis to Determine Buffer Overflow

Wagner et al. [26] 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 preconditions P1, P6, P11, and P15 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, so one cannot judge the potential precision of this technique based on the results in the paper.

4.3 STOBO

Haugh et al. [27] 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 run-time. Whenever an operation involving a buffer or string occurs, the appropriate constraints are checked using run-time 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 preconditions P1, P6, P11, and P15. 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.

4.4 Type-Assisted Dynamic Buffer Overflow Detection

Lhee and Chapin. [28] developed a technique that performs range checking on buffers that the program references. This is done at run-time, and requires a modification to the GNU C compiler. The method is to add a data structure 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 preconditions P1, P6, P11, and P15. 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 assumption 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.

4.5 CRED

The C Range Error Detector [29] implements a method of checking for out-of-bounds addresses, which 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 preconditions P1, P6, P11, and P15 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.

4.6 Jump Pointer Control

HSAP ("Hardware/Software Address Protection") [30] focuses on jump pointers, or locations that store an address to a code segment. Return addresses and function pointers are examples of jump 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 precondition P4. The system aborts the process.

HSAP handles function pointers using a special hardware register containing a random key. When a process begins, 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 precondition P6 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, precondition P17). It is not clear if this technique could be generalized to handle direct data buffer overflows. The problem is how the assignments to variables would be made when input occurred. 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.

4.7 StackGuard, MemGuard, and PointGuard

StackGuard [31] is a mechanism designed to thwart direct executable buffer overflows. It inserts a *canary*, or random number computed at run time, 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 precondition P3. It asserts that the return address has been changed, and therefore a buffer overflow occurs. If a buffer overflow attack was used to alter the return address, the attacker would need to overlay the canary with data containing he 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. Both are highly unlikely, but suggest an alternate approach.

Rather than generating a canary, a modified StackGuard approach would 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, rather than checking the canary, the process would check the value of the return address itself before executing the return. This eliminates false negatives, because if the return address has changed, precondition P3 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 from the current incarnation of StackGuard. Like StackGuard, this 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 could be used to thwart any of the buffer overflows described in this paper. It uses virtual memory protection mechanisms to protect specific memory locations such as the return address. MemGuard allows a precise implementation of preconditions P3, P8, P13, and P17, 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 [32] 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 preconditions P8 and P17, and suffers from the same problems as StackGuard.

4.8 SmashGuard

SmashGuard [33] provides micro-architectural support to detect attacks on the return address. It is a hardware-based approach that 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 return address on the process stack was altered and the program terminates.

This method handles direct executable buffer overflows only. It detects precondition P3 holding, and negates it by detecting the change to the return address. The discussion of countermeasures explains why this approach does not generalize well to the other three types of buffer overflow attacks. Further, the implementation of SmashGuard must keep the additional hardware stack in an area that the process cannot alter.

4.9 Split Control and Data Stack

This approach [34] is similar to that of SmashGuard, except that the implementation is presented in software as well as hardware. 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 precondition P3, and negates it when it is found to hold.

4.10 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 (RAS) used in modern processors. In normal operations RAS mis-predictions are the result of speculative updates of the stack and overflows due to limited RAS size. But buffer overflows can cause these mis-predictions. 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. But 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 technique also attempts to negate precondition P3 by detecting changes to the return address.

4.11 Non-Executable Stack

Several versions of the UNIX and Linux systems disable execute permission for the stack. This counters precondition P5, by preventing the program from executing instructions on the stack.

4.12 Summary

The focus of the techniques surveyed falls into negating (or detecting the existence of) four sets of preconditions. The first set is that the length of the input is longer than that of the buffer (preconditions P1, P6, P11, and P15). The second set is that a value—the return address or a function pointer—is altered (preconditions P3, P8, and P17). The third negates P5, the ability to execute instructions on the stack. The fourth detects branches into the stack (precondition P4). That leaves several other preconditions unexplored.

The difficulties in ensuring that preconditions P2, P7, P12, and P16 do not hold are discussed above. Strict typing that differentiates between data, addresses, and instructions is a step towards negating these preconditions. If *all* input is of type data, then the execution buffer overflows and the indirect data buffer overflows are eliminated because preconditions P2, P7, and P16 are false.

It is interesting that one technique (HSAP) checks for return addresses within the stack, but not for function pointers pointing to the heap or to data areas. A moment's thought explains the discrepancy. 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 this is not true for the heap, as 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 precondition P9 is typically not done. That dynamic loading uses the heap also means that detecting precondition P10 is equally useless.

Precondition P13 involves determining whether a value has been modified to cause an action that violates the security policy. Unfortunately, this is infeasible unless certain simplifying assumptions are made. 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.

Preconditions P14 and P18 are a function of the program, and basically 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 preconditions P14 and P18. As all programs involving security do rely on variables, it is infeasible to negate these preconditions taken alone.

5 Related Work

Cowan and his colleagues published an illuminating analysis of buffer overflow defenses [20]. They place defenses into four classes.

The first is *writing correct code*. This approach requires that programs be structured so that buffer overflow simply cannot occur. Range checking and careful management of pointers are necessary, and the goal of this approach is to ensure that the length of the data being loaded into the buffer will never

exceed the length of the buffer. Library functions and system calls must be carefully selected and invoked. This keeps preconditions P1, P6, P11, and P15 from holding, and therefore eliminates all buffer overflows.

The second is *non-executable buffers*. This approach makes the segments holding the data non-executable. This blocks the executable buffer overflows because it keeps preconditions P5 and P10 from holding.

The third class of defenses is *array bounds checking*. In this approach, a tool performs array and bounds checks similar to those that programmers writing correct code might perform. The tool can work on static code, like a bounds-checking compiler, or on dynamic, executing code. Hence, this approach is similar to the *writing correct code* approach and keeps the same set of preconditions from occurring.

The fourth class is *code pointer integrity checking*. This class of techniques examines pointers before they are dereferenced, and detects corruption. As Cowan et al. point out, this solves buffer overflow problems related to function pointers (specifically, both indirect buffer overflows, because it keeps preconditions P8 and P17 from holding).

6 Conclusion and Future Work

This paper provides a methodology for analyzing buffer overflow vulnerabilities. It derives specific conditions, called *preconditions*, that must hold in order for an attacker to exploit a buffer overflow vulnerability. These preconditions are both necessary and sufficient for the buffer overflows to be exploitable. Thus, if any precondition does not hold, a buffer overflow vulnerability does not exist.

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

The first approach is to examine the sets of preconditions, and focus on those common to all types of buffer overflows. The first common precondition is that the length of the input be no 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 common precondition is that the data being loaded must be treated as two different types: data (for the input) and either addresses or instructions (for the use). This precondition is harder to detect for the class of direct data buffer overflows, because the defense needs to distinguish between the type of the uploaded data and the type of the data in the variable. An approach using Biba's integrity model was briefly discussed.

The other preconditions 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 preconditions they negated.

Our preconditions 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 so. If the process is running with the attacker's privileges, an executable buffer overflow attack will gain the attacker nothing. An element of policy is therefore lacking in our preconditions. Incorporating this and other environmental elements into the preconditions in a way that an analyst can test would greatly enhance this work.

A second direction is the precise expression of the preconditions. Currently, they are stated in natural language. Ideally, they should be stated in a policy language to allow reasoning. This would enable us to tie the preconditions into the requires/provides attack models, and link vulnerabilities with attack models firmly. In the event that a site encode parts of its policy in such a language, 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 preconditions would provide an effective way of detecting potential security vulnerabilities. Preconditions underlie vul-

nerabilities, and many are common to more than one class of vulnerabilities. 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.

References

[1] Eichin, Mark W. and Jon A. Rochlis, "With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988," in *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 326–343, May 1989.

[2] Seeley, Donn, "A Tour of the Worm," in *Proceedings of the 1989 Winter USENIX Conference*, pages 287–304, January 1989.

[3] Spafford, Eugene H., "Crisis and Aftermath," *Communications of the ACM*, volume 32(6), pages 678–687, June 1989.

[4] CERT/CC, "W32/Blaster Worm," CERT Advisory CA-2003-20, CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 2003. Available online at http://www.cert.org/advisories/CA-2003-20.html.

[5] CERT/CC, "MS-SQL Server Worm," CERT Advisory CA-2003-04, CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, January 2003. Available online at http://www.cert.org/advisories/CA-2003-04.html.

[6] CERT/CC, "Apache/mod_ssl Worm," CERT Advisory CA-2002-27, CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 2002. Available online at http://www.cert.org/advisories/CA-2002-27.html.

[7] CERT/CC, "Continued Threat of the "Code Red" Worm," CERT Advisory CA-2001-23, CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, July 2001. Available online at http://www.cert.org/advisories/CA-2001-23.html.

[8] CERT/CC, ""Code Red" Worm Exploiting Buffer Overflow in IIS Indexing Service DLL," CERT Advisory CA-2001-19, CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, July 2001. Available online at http://www.cert.org/advisories/CA-2001-19.html.

[9] CERT/CC, "Buffer Overflow in Windows Workstation Service," CERT Advisory CA-2003-28, CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 2003. Available online at http://www.cert.org/advisories/CA-2003-28.html.

[10] CERT/CC, "Buffer Overflow in Sendmail," CERT Advisory CA-2003-25, CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 2003. Available online at http://www.cert.org/advisories/CA-2003-25.html.

[11] CERT/CC, "Buffer Overflow in Microsoft Windows HTML Conversion Library," CERT Advisory CA-2003-14, CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, July 2003. Available online at http://www.cert.org/advisories/CA-2003-14.html.

[12] CERT/CC, "Multiple Vulnerabilities in Snort Preprocessors," CERT Advisory CA-2003-13, CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 2003. Available online at http://www.cert.org/advisories/CA-2003-13.html.

[13] CERT/CC, "Buffer Overflow Vulnerability in Microsoft IIS 5.0," CERT Advisory CA-2001-10, CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 2001. Available online at http://www.cert.org/advisories/CA-2001-10.html.

[14] CERT/CC, "Multiple Buffer Overflows in Kerberos Authenticated Services," CERT Advisory CA-2000-06, CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 2000. Available online at http://www.cert.org/advisories/CA-2000-06.html.

[15] Abbott, R. P., J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb, "Security Analysis and Enhancements of Computer Operating Systems," Technical Report NBSIR 76-1041, Institute for Computer Sciences and Technology, National Bureau of Standards, April 1976.

[16] Aslam, Taimur, "A Taxonomy of Security Faults in the UNIX Operating System," Master's Thesis, COAST Technical Report 95-09, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, August 1995.

[17] Bisbey II, Richard and Dennis Hollingworth, "Protection Analysis: Final Report," Technical Report ISI/SR-78-13, University of Southern California Information Sciences Institute, Marina Del Rey, California, May 1978.

[18] Landwehr, Carl E., Alan R. Bull, John P. McDermott, and William S. Choi, "A Taxonomy of Computer Program Security Flaws," *ACM Computing Surveys*, volume 26(3), pages 211–254, September 1994.

[19] Bishop, Matt, "Vulnerability Analysis," in *Proceedings of the Second International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 125–136, September 1999.

[20] Cowan, Crispin, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," *DARPA Information Survivability Conference and Exposition*, volume 2, page 1119, January 2000.

[21] Aleph One, "Smashing The Stack For Fun And Profit," *Phrack Magazine*, volume 7(49), August 1996.

[22] Conover, Matt, "w00w00 on Heap Overflows," w00w00 Security Development (WSD), January 1999. Available online at http://www.w00w00.org/files/articles/heaptut.txt.

[23] Esser, Stefan, "Samba 3.x QFILEPATHINFO Unicode Filename Buffer Overflow," Advisory 13/2004, e-matters, November 2004.

[24] Newsham, Tim, "Format String Attacks," Guardent, Incorporated, September 2000. Available online at http://www.thenewsh.com/~newsham/format-string-attacks.pdf.

[25] Biba, Kenneth J., "Integrity Considerations for Secure Computer Systems," Technical Report MTR-3153, The MITRE Corporation, Bedford, MA, June 1977.

[26] Wagner, David, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," in *Proceedings of the 2000 Network and Distributed System Security Symposium*, pages 1–15, February 2000.

[27] Haugh, Eric and Matt Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," in *Proceedings of the 2003 Symposium on Networked and Distributed System Security*, February 2003.

[28] Lhee, Kyung-Suk and Steve J. Chapin, "Buffer Overflow and Format String Overflow Vulnerabilities," *Software—Practice & Experience*, volume 33(5), pages 423–460, April 2003.

[29] Ruwase, Olatunji and Monica S. Lam, "A Practical Dymanic Buffer Overflow Detector," in *Proceedings of the 2004 Symposium on Network and Distributed System Security*, pages 159–169, February 2004.

[30] Shao, Zili, Qingfeng Zhuge, Yi He, and Edwin Sha, "Defending Embedded Systems Against Buffer Overflow via Hardware/Software," in *Proceedings of the 19th Annual Computer Security Applications Conference*, page 352, December 2003.

[31] Cowan, Crispin, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium*, page 5, USENIX Association, Berkeley, California, 1998.

[32] Cowan, Crispin, Steve Beattie, John Johansen, and Perry Wagle, "PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities," in *Proceedings of the 12th Conference on USENIX Security Symposium*, pages 91–104, USENIX Association, Berkeley, California, August 2003.

[33] Özdoganoglu, Hilmi, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote, "SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address," *IEEE Transactions on Computers*, volume 55(10), pages 1271–1285, October 2006.

[34] Xu, Jun, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer, "Architecture Support for Defending Against Buffer Overflow Attacks," in *Proceedings of the Workshop on Evaluating and Architecting System Dependability*, 2002.