# Buffer-Overflow Protection: The Theory

Krerk Piromsopa, *Member, IEEE*, and Richard J. Enbody, *Member, IEEE*

*Abstract*— **We propose a framework for protecting against buffer overflow attacks—the oldest and most pervasive attack technique. The malicious nature of buffer-overflow attacks is the use of external data (input) as addresses (or control data). With this observation, we establish a sufficient condition for preventing buffer-overflow attacks and prove that it creates a secure system with respect to buffer-overflow attacks. The underlying concept is that input is untrustworthy, and should not be use as addresses (return addresses and function pointers.). If input can be identified, buffer-overflow attacks can be caught. We used this framework to create an effective, hardware, buffer-overflow prevention tool.**

***Index Terms***—**Buffer overflow, Buffer-Overflow Attacks, Computer security, Function-Pointer Attacks, Intrusion Detection, Intrusion Prevention**

## I. Introduction

We create a theoretical foundation for a secure system with respect to buffer-overflow attacks. The goal is to use the foundation to provide a framework for implementing buffer-overflow protection.

Although they date back to the infamous MORRIS worm of 1988 [3], buffer-overflow attacks remain the most common. Though skilled programmers should write code without buffer overflows, no program is guaranteed free from bugs so it cannot be considered completely secure against buffer-overflow attacks. The persistence of buffer-overflow vulnerabilities speaks to the difficulty of eliminating them. In addition, as buffer overflow vulnerabilities are eliminated in operating systems, they are being found and exploited in applications. When applications are run with root or administrator privileges the impact of a buffer overflow is equally devastating.

In an effort to avoid relying on individual programming skill, a number of researchers have proposed a variety of methods to protect systems from buffer-overflow attacks. Most of them are not able to provide complete protection. For example, some only prevent the original stack-smashing attack, but can be circumvented by more recent attacks.

It is worth clarifying that buffer-overflow attacks can be done on any data and variables. This paper focuses on buffer-overflow attacks on control data, but is not limited to control data.

K. Piromsopa is a Ph.D. student in the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA (corresponding author to provide phone: 517-353-3148; fax: 517-432-1061; e-mail: piromsop@cse.msu.edu).

R. J. Enbody is with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA. (e-mail: enbody@cse.msu.edu).

The goal of this paper is to provide a generic framework for preventing buffer-overflow attacks. We begin by defining buffer overflows in general. Next, we establish a sufficient condition for preventing buffer-overflow attacks and prove that it will create a secure system with respect to buffer-overflow attacks.

## II. Background

This section is intended to be a gentle introduction to buffer overflows

### *A. Fundamental of Buffer-Overflow Attacks*

First, we define buffer overflow(from the Webopedia [18]).

***Definition 1:***

A **buffer overflow** is the condition wherein the data transferred to a buffer exceeds the storage capacity of the buffer and some of the data "overflows" into another buffer, one that the data was not intended to go into.

Since buffers can only hold a specific amount of data, when that capacity has been reached the data has to flow somewhere else, typically into another buffer, which can corrupt data already in that buffer.

Exploiting buffer overflow can lead to a serious system security breach (buffer-overflow attack) when necessary conditions are met. The seriousness of buffer-overflow attacks ranges from writing into another variable, another process' memory (segmentation fault), or redirecting the program flow to execute malicious or unexpected code. Based on the definition of buffer overflow, Definition 2 defines buffer-overflow attacks.

***Definition 2:***

A **buffer-overflow attack** is an attack that (possibly implicitly) uses memory-manipulating operations to overflow a buffer which results in the modification of an address to point to malicious or unexpected code.

In general, a buffer-overflow attack is an attack on any data (including variables and addresses). To make this paper readable, the term "buffer-overflow attacks" is used to refer to attacks on control data.

***Observation:*** An analysis of buffer-overflow attacks indicates that a buffer of a process is always overflowed with a buffer passed from another domain (machine, process)—hence its malicious nature.

***Assumption 1:***
In an attack, a buffer is always overflowed using a buffer passed from another domain.

This concept is not new. For example, Howard and LeBlanc state in their book "All input is evil until proven otherwise" [9]. Accordingly, an intuitive way to prevent buffer-overflow attacks is to detect and validate input, especially input which is eventually used for control.

In early attacks, the attacked address was a return address, but later other control data (e.g. function pointers, jump table) were attacked. In all cases, the eventual access of that address (e.g. by a return, function call or jump) will redirect the program control flow to execute the malicious or unexpected code. If the address was modified by something other than a buffer overflow, it is a race condition, a Trojan horse, or other type of attack.

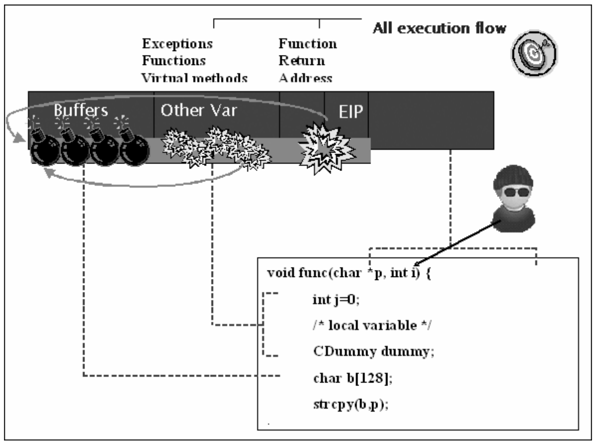
.

## *B. Sample Attacks and Variations*

There are two main targets of buffer-overflow attacks: control data and local variables. In the vast majority of attacks, control data is the target so prevention schemes have focused on control data. Control data can be divided into several types: return addresses, function pointers, and branch slots. Return addresses have been the primary target since their location can easily be guessed. More advanced buffer-overflow attacks target other control data. Some literature refers to attacks on return addresses as first-generation attacks, and those on function pointers as second-generation attacks [5].

The first step-by-step description of how to construct a buffer-overflow attack was written by Elias Levy (a.k.a. Aleph One) in 1996 [11]. He first used the term "Stack Smashing" to refer to plastering the stack with shell code and its address to set up an eventual overflow of a return address on the stack.

Figure 1 shows a stack-smashing example. In this example, an attacker (shown in the black hat) passes a buffer containing malicious code (e.g. shell) and multiple copies of the address of the target buffer as an argument to the vulnerable program (through parameter *p*). A buffer manipulation function (e.g. strcpy in this example) will overflow the function's return address with the address of the target buffer containing malicious code. The eventual result is that the return instruction will use the address of the target buffer as a return address and return the program flow to execute that malicious code. Additional attack vectors are provided (but not used) in the code such as function pointer *fp*. As outlined above, it is important to note that the critical component of this attack is the modification of the return address (replaced by the address of the target buffer).

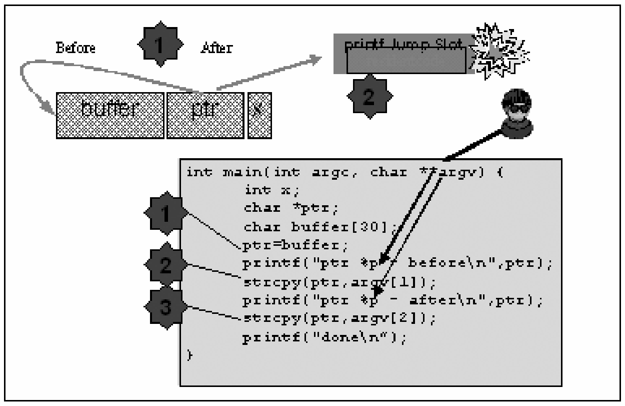


```
void func(char *p, int i) {
        int j=0;
        /* local variable */
        CDummy dummy;
        char b[128];
        strcpy(b,p);
```

**Figure 1 Stack Smashing**

To illustrate an advanced buffer-overflow attack, we provide a multistage buffer-overflow attack [10] (a.k.a. Hannibal Exploit [6]) that can bypass most software buffer-overflow solutions. Fundamental to a multistage buffer-overflow attack is that there exists a vulnerable pointer to a buffer. That is, there is a user-writeable buffer sufficiently near a useful pointer. First, the pointer is modified (by overflowing) to point to a specific location (e.g. a jump slot or a function pointer). In the second stage of the attack, an input is stored at the pointer's target. These two steps allow attackers to create a pointer to any location (first stage) and overwrite the pointer's target with a desired value (second stage). The next time some program jumps to that target, it will be redirected based on the value inserted by the attacker. In particular the program will be redirected to the attacker's malicious code. For example, if the program is running in privileged mode and the pointer points to shell code, the attacker will have created a privileged shell allowing free reign. Figure 2 is an example of such a vulnerable program.

Before examining the code, let's review how a jump table is used. Consider the slot in the table for the pointer to the *printf* executable. A call to *printf* indexes to that slot in the table and then jumps to the *printf* executable. To attack this type of program, the buffer-overflow is done in two stages. First, the *ptr* pointer is overflowed to point to a desired memory location (1), e.g. the *printf* slot in the jump table. In particular, *argv[1]* controlled by the attacker will contain the address of the *printf* slot in the jump table. The strcpy routine will copy *argv[1]* into *buffer,* but overflows to overwrite *ptr* with the *printf* address slot. In the figure, we see that the pointer *ptr* originally pointed to 'buffer' (arc labeled 'Before') but now points to the jump slot (arc labeled 'After). Now that *ptr* points to the *printf* slot in the jump table, we need to insert a desired value into that slot. Suppose, for illustration, that we also have determined the address of resident shell code (we'll call it *residentcode*). Using our modified *ptr* we will overwrite the jump table slot with the *residentcode* address. We use the second strcpy call (2) to write *argv[2]* (also controlled by the attacker and whose value is *residentcode*) into the target of *ptr* which now points to the *printf* entry in the jump table. The result of that second strcpy call is that we have placed the address *residentcode* (resident shell code) into the *printf* slot of the jump table. The attacker has achieved his goal. Now when a program calls *printf,* control passes as usual to the *printf* entry in the jump table, but now the attacker has redirected control to *residentcode,* the address of the shell code. Instead of *printf* a shell will be started. If the program which called *printf* was operating in privileged mode, the attacker will have succeeded in creating a privileged shell with full system access. (See [13] for more details). As we will see below, this multi-stage attack gets around most buffer-overflow protection schemes. Less obvious is that we can use a similar approach to circumvent some software solutions to buffer-overflow attacks by modifying a handling vector which can allow us to bypass the buffer-overflow handling routine.

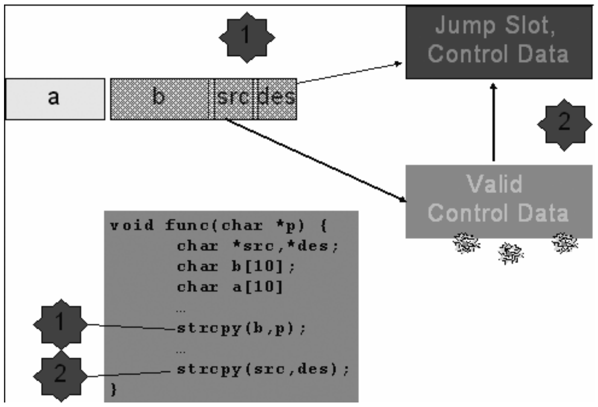


```
int main(int argc, char *argv) {
        int x;
        char *ptr;
        char buffer[30];
        ptr=buffer;
        printf("ptr *p - before\n",ptr);
        strcpy(ptr,argv[1]);
        printf("ptr *p - after\n",ptr);
        strcpy(ptr,argv[2]);
        printf("done\n");
}
```

**Figure 2 An example of a vulnerable program**

Another variation which most schemes cannot protect is an attack on generic pointers. By modifying both the source and destination pointers, copying from one arbitrary memory location to another is possible. Figure 3 illustrates an attack based on attacking pointers. In this figure, the first ***strcpy*** will allow attackers to overflow buffer ***b*** so the ***src*** and ***des*** pointers are replaced with two pointers of the attacker's choice: in this case, valid control data and target address respectively. The second ***strcpy*** will then copy from the ***src*** location to the ***des*** location. Using this approach, it is possible to modify any memory entry with a known local entry while avoiding most protection mechanisms. For example, an encoded function pointer can be used to attack another function pointer (e.g. jump slot).

A related attack is worth noting here: printf vulnerabilities [17]. Malformed formatting instructions can allow arbitrary memory to be overwritten. It is not a buffer-overflow attack, but with the ability to overwrite arbitrary memory the attack then proceeds like many buffer-overflow attacks by attacking control data. Some buffer-overflow schemes can prevent some of those attacks. However, any variable can also be attacked, and no buffer-overflow scheme protects arbitrary data. Fortunately, simple static analysis of source code can identify printf vulnerabilities. As we will see below, static analysis is not sufficient for buffer overflows.



```
void func(char *p) {
        char *src,*des;
        char b[10];
        char a[10]
        ...
        strcpy(b,p);
        ...
        strcpy(src,des);
}
```

**Figure 3 Buffer-Overflow Attacks on Pointers**

Recently "integer overflows" (known more generically as "integer arithmetic" attacks) have emerged as a variation [2]. In these attacks guard data which protects buffers is attacked. After defeating the guard data some buffer-overflow attack is used. Therefore, these attacks can be considered as a variation of a buffer-overflow attack which the more robust schemes can protect against.

An attack on local variables is exemplified by the classic password attack from a 1987 paper [19]. Basically, a variable is being overflowed allowing an arbitrary password to be validated resulting in root or administrator access. No control data is involved so no existing buffer-overflow protection schemes protect against this type of attack.

## III. PREVENTION

This section discusses the necessary conditions for preventing buffer-overflow attacks on control data.

***Postulate 1:***

In buffer-overflow attacks on control data, the generic buffer/memory-manipulating operations are used by the vulnerable routine to overflow the address (e.g. a return address or a function pointer).

From Definition 2, we observe that preserving the integrity of the address is a sufficient condition to prevent this class of buffer-overflow attacks. To clarify, Definition 3 shows the meaning of the integrity of an address in this context.

***Definition 3:***

Maintaining the integrity of an address means that the address has not been modified by overflowing with a buffer passed from another domain.

Consider the implication of Definition 3 in light of our "Observation" about Definition 2 which noted the importance of attacks working across domains (machines, processes): in order to preserve the integrity of the address (e.g. a return address or a function pointer), an address cannot be created from data passed across domains (e.g. machines, processes) via buffer overflow.

To maintain its integrity, the address created locally can be signed when it is created and is validated by associated instructions (e.g. return, call, and jump instructions) before they are completely executed. Implicitly, a signature represents some metadata associated with the address. Necessarily, the signature must not be passed across domains. If the signature could be passed across domains, a valid address could be used for attacking a system. If we assume that a signature only exists locally, the last condition is enforced when a buffer is passed across a network/hardware device where the signature cannot be passed.

If local data can be differentiated from data passed from another domain, we can detect buffer-overflow attacks on control data. Thus we may reverse the signature by signing data that passed across domains and leave the local data unsigned. This scheme provides better backward-compatibility, since no modification is required for legacy processes.

With these definitions, Theorem 1, and its corollary are introduced. The corollary is the key to the entire framework presented in this paper since it defines a *sufficient* condition for buffer-overflow attacks.

***Theorem 1:***
Modifying an address by replacing ("overflowing") it using a buffer passed from another domain is a *necessary* condition for a buffer-overflow attack on control data.

***Restatement:*** If there is to be a buffer-overflow attack on control data, an address must be modified using a buffer passed from another domain.

***Proof:***
Theorem 1 follows directly from Definitions 1 and 2.

***QED***

***Corollary 1.1:***
Preserving the integrity of an address is a *sufficient* condition for preventing a buffer-overflow attack.

***Restatement:*** If the integrity of an address is preserved, that is a sufficient condition for preventing a buffer-overflow attack.

***Proof:***
From Theorem 1, "If there is to be a buffer-overflow attack, an address must be modified by manipulating a buffer from another domain." The contrapositive of that statement is "If an address cannot be modified (or such modification can be detected), then a buffer-overflow attack is not possible." We know that the contrapositive of a true statement is true.

***QED***

Intuitively, from Definition 2, the attack is the ability to redirect the program flow to execute malicious or unexpected code. To achieve this goal, the address must be modified. If the address cannot be modified, the buffer-overflow attack fails. If modification of the address can be recognized, the buffer-overflow attack can be recognized and stopped. On the other hand, if the address can be validated, execution can proceed safely.

## IV. Secure System

To claim that this framework can enforce the integrity of the addresses and result in a secure system, a validation will be discussed. Assuming that a computer system can be represented as a finite-state automation with a set of transition functions, we can define a secure system (with Definitions 4 and 5).

***Definition 4:***
A **security policy** is a statement that partitions the states of the system into a set of authorized, or secure, states and a set of unauthorized, or insecure, states [1].

In the case of buffer-overflow attacks, the security policy is simply the statement:

***Protocol 1:***
Overflowing a buffer cannot create a valid address
(e.g. a return address or a function pointer)

which follows from Corollary 1.1. Before going further, we first define a secure system.

***Definition 5:***
A **secure system** is a system that starts in an authorized state and cannot enter an unauthorized state. [1].

***Theorem 2:***
A system which preserves the integrity of an address (e.g. a return addresses or a function pointer) is a secure system with respect to buffer-overflow attacks.

***Restatement:*** A system that does not use input as a control data is a secure system with respect to buffer-overflow attacks on control data.

***Proof:***
Assume that a system is partitioned into two states: normal operation and buffer-overflow attack. By the definition of buffer-overflow attacks (Definition 2), only overwriting the address (e.g. a return address or a function pointer) with an address passed as a buffer (input) to vulnerable programs will result in the state of buffer-overflow attack. By the definition of preservation of the address (Definition 3), if such overflowing can be recognized and prevented, the system will not result in the state of buffer-overflow attacks. With respect to Definition 5, our system cannot enter an unauthorized state and is considered to be a secure system.

***QED***

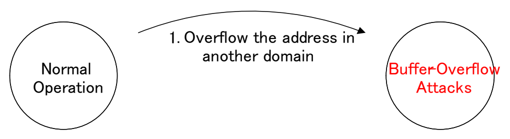


Figure 4 State diagram of buffer-overflow attacks.

Hitherto, we will show that the enforcement of Protocol 1 (stated early) results in a secure system with respect to buffer-overflow attacks.

***Theorem 3:***

A system that enforces Protocol 1 can preserve the integrity of an address, and result in a secure system with respect to buffer-overflow attacks.

***Proof:***

By enforcing Protocol 1, we can detect that an address (e.g. a return address or a function pointer) is overflowed by a buffer passed from another domain (including input). If we can detect that an address is modified by a buffer from another domain, we can preserve the integrity of the address. This follows directly from Definition 3. Thus a system that can enforce Protocol 1 preserves the integrity of the address and is a secure system with respect to buffer-overflow attacks. This follows directly from Theorem 2

***QED***

## V. Implementations

It was suggested that metadata is a key to preventing buffer-overflow attacks in a 2003 article [8]. There exist at least four methods that follow our proposed framework by using metadata for tracking input. These methods are Minos [6] [7], Tainted Pointer [3], Dynamic Flow Tracking [16] and Secure Bit [14]. These methods share the same assumption that input should not be used as control data.

All methods required an additional bit augmented to each memory word (or byte). This bit is used for tracking input. Whenever input is passed to a process, the bit is marked. Upon execution, Jump, Call or Return instructions validate that control data is not derived from input. If input is about to be used as an address, an exception is raised.

However, the implementations differ. Minos views data across segments as input. Tainted Pointer considers data passed from the operating system as input. Dynamic Flow Tracking relies on operating systems for marking input. Secure Bit treats data passing between processes through the kernel as input. More details of buffer-overflow attacks and protection schemes can be found in [12] and [15] respectively.

## VI. Summary

Buffer-overflow attacks on control data require overflowing addresses (return addresses and function pointers) with a buffer passed from another domain (machine, and process). In this paper we developed a formal argument that "a necessary condition for preventing buffer-overflow attacks is the preservation of the integrity of addresses across domains". We then show how a protocol based on that statement supports a variety of successful hardware-based methods to prevent buffer overflow attacks. Our formalism lends credence to their claims of success.

.

## References

[1] M. Bishop, Computer Security, Addison-Wesley, (Dec. 2002)
[2] Blexim, "Basic Integer Overflow," 2002. Available: http://www.phrack.org/phrack/60/p60-0x0a.txt
[3] C. Schmidt, and T. Darby, "The What, Why, and How of the 1988 Internet Worm," http://www.snowplow.org/tom/worm/worm.html
[4] S. Chen, J. Xu, N. Nakka, Z. KalbarcZyk, and R. K. Iyer, "Defeating Memory Corruption Attacks via Pointer Taintedness Detection," *Proc. Of IEEE International Conf. on Dependable Systems and Networks (DSN)*, Yokohama, Japan, June 28 - July 1, 2005
[5] E. Chien, and P. Szor, "Blended Attacks Exploits, Vulnerabilities and Buffer-Overflow Techniques in Computer Viruses," *Proc. of Virus Bulletin Conf*, 2002
[6] J.R. Crandall, and F.T. Chong, "Minos: Control Data Attack Prevention Orthogonal to Memory Model," *Intl. Sym. on Microarchitecture*, 2004.
[7] J.R. Crandall, and F.T. Chong, "A Security Assessment of the Minos Architecture," *ACM SIGARCH, Vol 33. No. 1*, 2005.
[8] A. Glew, "Segments, Capabilities, and Buffer Overrun Attacks," Computer Architecture NEWS, ACM SIG Computer Architecture Vol.31, No.4 - September 2003, pp. 26 – 31
[9] M. Howard, D. Leblance, Chapter 10:All Input Is Evil!. Writing Secure Code, Microsoft Press, 2nd ed.(1965)
[10] S. Hsiangren, "Apache/mod_ssl (slapper) Worm," GIAC Certified Incident Handler, 2002
[11] A. One, "Smashing stack for fun and benefit," *Phrack Mag, 49(7)*, 1996.
[12] J. Pincus, and B. Baker, "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns," IEEE Security & Privacy, Vol. 2, No. 4, July/August 2004, pp. 20 - 27
[13] K. Piromsopa, and R. Enbody, "Buffer Overflow: Fundamental," Technical Report #MSU-CSE-04-47, Dept. of Computer Science and Engineering, Michigan State University, 2004.
[14] K. Piromsopa, and R. Enbody, "Secure Bit2 : Transparent, Hardware Buffer-Overflow Protection," Tech.Report #MSU-CSE-05-9, Dept of Computer Science and Engineering, Michigan State University, 2005.
[15] K. Piromsopa, and R. Enbody. "Survey of Buffer-Overflow Protection,"Technical Reports #MSU-CSE-06-3, Department of Computer Science and Engineering, Michigan State University, 2006.
[16] G. Suh, J. Lee, S. Devadas, "Secure program execution via dynamic information flow tracking," In *ASPLOS XI* (Oct, 2004.)
[17] U. Shankar, K. Talway, J.S. Foster, and D. Wagner, "Detecting Format String Vulnerabilities with Type Qualifiers," In *Proc. of the 10th USENIX Security Symposium*
[18] Webopedia. What is buffer overflow?, http://www.webopedia.com/TERM/B/buffer_overflow.html
[19] W.D. Young, "Coding for a Believable Specification to Implementation Mapping," *IEEE Symp on Security and Privacy* 1987: pp. 140-149.