Defeating Buffer-Overflow Prevention Hardware

Krerk Piromsopa
Department of Computer Science and Engineering,
Michigan State University,
East Lansing, MI 48824 USA
1-517-353-3148
piromsop@cse.msu.edu

Richard J. Enbody
Department of Computer Science and Engineering,
Michigan State University
East Lansing, MI 48823 USA
1-517-353-3148
enbody@cse.msu.edu

ABSTRACT
Buffer overflow attacks persist in spite of advances in software engineering. Numerous prevention schemes in software have been developed over the years, but so have techniques to circumvent them. Recently, improved schemes have appeared which are entirely in hardware or require hardware modifications to support them. In this paper we describe how to defeat or circumvent these improved mechanisms. Included are well-known attacks such as the Hannibal attack (a multistage buffer-overflow attack) as well as a new attack which we call the "arbitrary copy" attack—an attack specifically designed to defeat the hardware approaches. To aid the reader, the hardware prevention mechanisms will be described.

Categories and Subject Descriptors
K.6.5 [**Management of Computing and Information Systems**]: Security and Protection – *invasive software, physical security, unauthorized access*

General Terms
Management, Reliability, Security, Theory

Keywords
Buffer overflow, Buffer-Overflow Attacks, Computer security, Intrusion Detection, Intrusion Prevention

1. INTRODUCTION
In this paper, we will present two types of buffer-overflow attacks that are able to bypass most buffer-overflow protection mechanisms, especially hardware mechanisms which are the most effective. We refer to these attacks as “multistage buffer-overflow attack”[4], and “arbitrary copy”. The multistage attack is an attack that allows attackers to create an arbitrary pointer and modify any data it points to. Arbitrary copy is an attack on two data pointers that allows an attacker to copy data from one arbitrary location to another arbitrary location. Either attack can result in a serious compromise of a system.

WDDD'06, June 18, 2006, Boston, MA., USA.

The goal of this paper is to provide a rudimentary understanding of the current state-of-the-art solutions against buffer-overflow attacks (the best are hardware mechanisms) and then show how to circumvent these mechanisms. We begin by examining the the attacks and their potential threat. Then we look at current protection schemes and the impact of the attacks on them.

2. BACKGROUND
This section begins by reviewing the characteristics of buffer-overflow vulnerabilities and attacks. Later we briefly analyze current solutions against buffer-overflow attacks. In particular, we will focus on a promising scheme, namely input protection.

2.1 Buffer-Overflow Attacks
Although, they date back to the infamous MORRIS worm of 1988 [24], 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, so they can be circumvented by more recent attacks.

Buffer-overflow attacks occur when a malformed input is being used to overflow a buffer causing a malicious or unexpected result. Some metadata is necessary for prevention [11].

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 [3].

3. MULTISTAGE BUFFER-OVERFLOW ATTACK

Multistage buffer-overflow attack (a.k.a. Hannibal Exploit [4]) refers to a type of attack that requires several steps of buffer overflow. This type of attack is able to bypass most protection schemes such as Address Protection, Bounds checking, and Obfuscation. 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 1 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 (Figure 1). 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.

The Multistage attack seems convoluted, but it is the technique used by the infamous Slapper [31] worm. Most protection techniques focus on protecting the stack, the return address on the stack, or the heap. Neither the stack nor the heap is involved in this attack so all those techniques were ineffective against the Slapper worm.

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.

4. ARBITRARY COPY ATTACK

There exists an arbitrary copy primitive which may allow attackers to modify control flow without using external control data. This type of attack is more advanced in that it can bypass protection schemes which can catch the multistage attack described above. Using strcpy one can construct a vulnerable routine such that using a buffer-overflow to modify source and destination pointers, an attacker can arbitrarily copy any data from one location to another. This technique allows an existing piece of control data to overwrite another piece of control data. The result is control flow other than what the original programmer

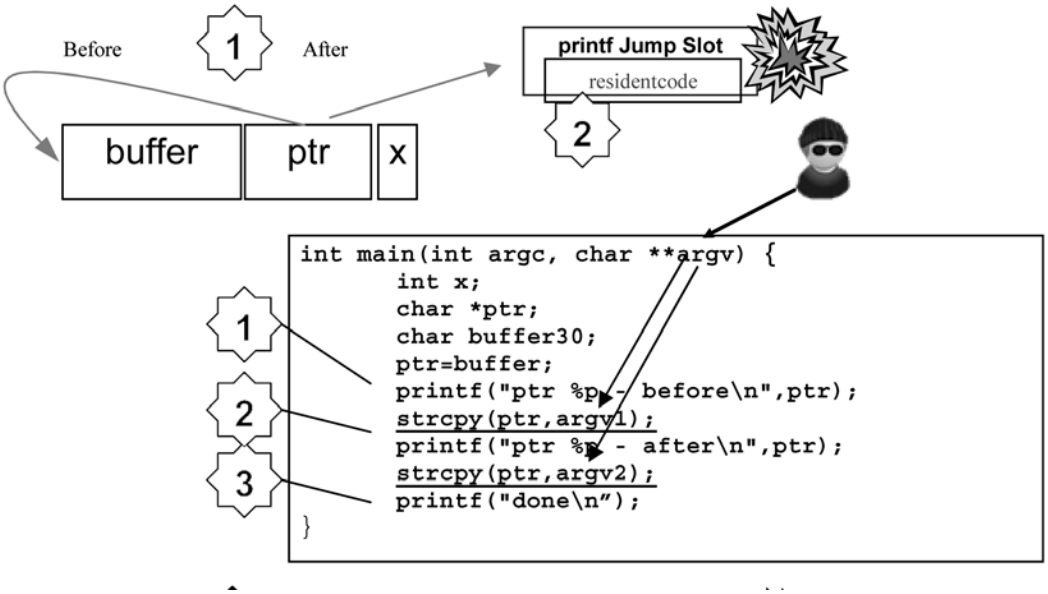

Figure 1 Multistage Buffer-Overflow Attacks

intended. Necessary conditions for the success of this type of attack are:

- A vulnerable copy function such that a user can modify both arguments (source and destination pointers) (possibly using buffer-overflow attacks) as exemplified in Figure 2.
- The (useful) control data is stored in the local memory area.

```
char *src,*dest;
char buff[10];

gets(buff);
...
strcpy(src,dest);
```

Figure 2. Vulnerable code for arbitrary copy attack

Both of these conditions must be true. If one fails, the attack fails. Though the first condition could be satisfied in any arbitrarily program, the code generated by the compiler could render the attack impossible. For example, any level of optimization should use registers for storing the source and destination variables. If either or both are in registers, a buffer-overflow to modify both variables will fail. We will analyze the possible cases where both conditions concurrently occur later.

4.1 Example

To ease understanding, Figure 3 presents a sample case of an attack on non-control data where the vulnerability might be applicable.

In this example, main calls function "*a*" which then calls the vulnerable function "*b*". Within "*b*" the user inputs *buff* which can overflow to both overwrite **src* to point to the return address of a previous call (e.g. "*a()*") and overwrite **dest* to point to the target address (e.g. return address of "*b()*" or "*main()*"). Note that this overflow is possible only if all optimization is turned off so that neither *src* nor *dest* is in a register. Under these circumstances it is possible to change the control flow without replacing control data with external data—only internal data is used. Note that the damage in this example is to create an infinite loop or crash the program, effectively a denial of service to the process.

While most internal data targets will be benign, one can imagine malicious possibilities. For example, if for some reason a programmer created a function pointer to *shell* and had both a vulnerable copy routine and no optimization; one could copy that shell pointer elsewhere to allow a shell call someplace different than the programmer intended. Note that the desired privileged-elevated shell is not possible with this attack because the best buffer-overflow prevention schemes will prevent privilege-elevation attacks. Alternatively, (again with a vulnerable copy routine and no optimization) if one had function pointers to both an authorization "accept" function and a "reject" function one might be able to redirect program flow to subvert an authorization routine to the "accept" function when the "reject" function was expected. Figure 4 shows a possible scenario.

```
int b() {
     char *src,*dest;
     char buff[10];
     printf("Input string:.\n");
// Overflow *src, *dest
     gets(buff);
// Copy src to dest
     strcpy(src,dest);
}

int a() {
     …
     b();
     …
}

int main (int argc,char *argv[])
{
     a();
}
```

Figure 3 Sample Buffer-Overflow attacks on non-control data

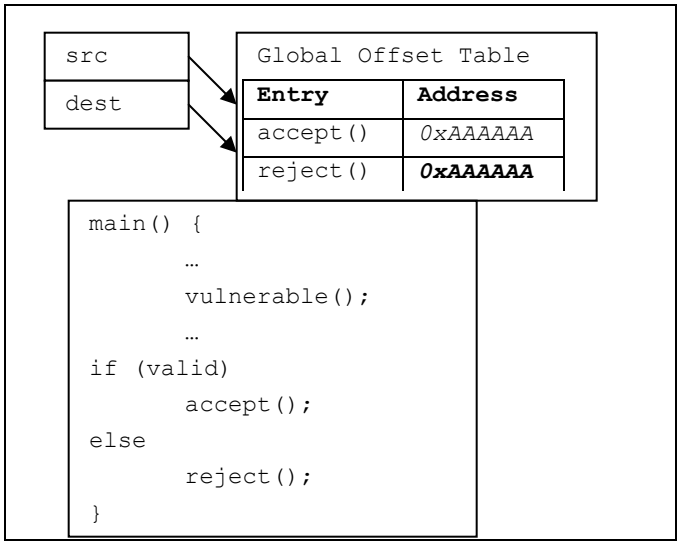

Figure 4 a possible scenario

4.2 Issues

Consider the second condition of arbitrary copy: the presence of a useful address in local memory. We know that input protection mechanisms prevent the use of input as control data, thus only purely local data that is not derived from input is a potential threat. One's first thought might be that any function call could provide an address of that function. However, because of relocation, local calls use relative addresses which cannot be used for this attack. Other sources of control targets such as jumps are also relative addresses and not useful. Given this observation, potential sources of addresses are narrowed to the presence of a shared library or a function pointer.

Shared library. In the case of shared libraries (the function is located in the shared library), a call to the function means there exists a useful entry in the Global Offset Table (GOT).

Function Pointer. The assignment of a function address to a function pointer (frequently found in C++) would create a pointer available for reuse.

If a useful address is stored as an entry in the GOT or a function pointer, the buffer-overflow described above can be used to replace a target address with this address. Target address might be return addresses, function pointers, or an entry in GOT itself.

The probability that all conditions are applicable is considered to be low. In fact, some researchers [4] do not believe that it will be a problem or suggest that encoding addresses in GOT should be sufficient for preventing the attack. However, that prevention might not be able to protect some function pointers in C++.

We have to eliminate at least one critical condition. There are three possible methods.

- Prevent a raw address from being stored directly in the program.
- Secure the target address from being modified (e.g. GOT and function pointers).
- Validate that both the source and destination pointer have not been maliciously modified.

Rather than storing an address directly into the GOT table or function pointer, we may choose to store an encoded version of an address or store a relative address. Even a trivial encoding such as XOR (like PointGuard [7]) with some constant would be sufficient. However, this approach does not prevent a copy between locations that share the same encoding scheme or key used to encrypt the address (e.g. between function pointers or entries in the GOT). Note that PointGuard [7] can be used to reduce the probability of overwriting source and destination pointers. However, if the key and algorithm can be circumvented, it is possible to overwrite it with a valid copy. In fact, we may be able to overflow the value (e.g. index) that is used for pointer arithmetic rather than modifying the pointer directly.

Rather than making the useful address useless, we can protect the target from being modified. In the case of GOT, we can protect the GOT from being a target by declaring it as read only after the shared library is configured. Nonetheless, we cannot apply the same idea to protect function pointers or return addresses in general.

Alternatively, we can validate (assert) the source and destination pointers before running the "strcpy(..)" function. If the source and destination pointers can be validated, the attack can be prevented. However, a false alarm may be generated when a pointer is the arithmetic result of input.

5. CURRENT PROTECTION SCHEMES

Current approaches against buffer-overflow attacks include both software and hardware techniques and can be partitioned into three broad categories: static analysis, dynamic solutions, and isolation. We focus on the hardware mechanisms, but mention the software solutions for completeness. Static analysis is a software methodology which tries to fix functions that are vulnerable to buffer-overflow attacks. Dynamic approaches can be entirely in software, entirely in hardware or some combination. Dynamic techniques monitor or protect data that is either a target or the source of buffer-overflow attacks. Isolation seeks to limit the damage of attacks and attempts a broader protection than simply buffer overflow. The best-known isolation techniques are in hardware. To ease understanding, Figure 5 shows taxonomy of protection schemes. We will base our discussion on this classification.

The main idea of "Static analysis" is to find and solve the problem before deploying a program. To do so, we first analyze the source code or disassembly of the program by looking for code with a predefined signature. Examples of tools in this category are: ITS4 [29], FlawFinder [9], RATS [25], and STOBO [12]

Isolation schemes isolate the attacker either to eliminate an attack vector or to contain damage after a successful attack. Preventing the execution of code in stack memory isolates the stack from the attacker. Alternatively, limiting the memory of a process can isolate a compromised process. Non-executable memory is an example of the former while sandboxing is an example of the latter. Examples include AMD NX [17], non-executable stack [27], SPEF [16], and sandboxing.

Knowing which data are critical to attacks, we can prevent attacks by validating the integrity of that data. As mentioned above, the data of interest are control data such as (but not limited to) return addresses. We name these "Dynamic Solutions" because data are dynamically managed and verified in the run-time environment. The most effective prevention mechanisms fall in this category.

In a survey of buffer-overflow protection [23], it was suggested that metadata is necessary for validating the integrity of data. While the assumptions of critical data and the methods for storing and validating metadata vary from one solution to another, dynamic solutions can be classified into four groups:

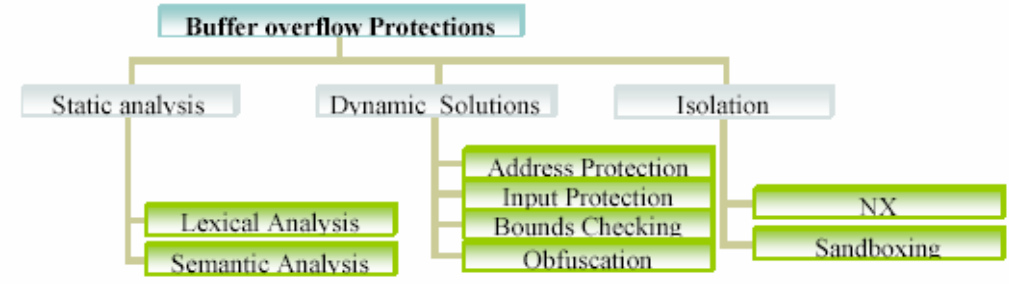

Figure 5 Taxonomy of protection schemes

- Address Protection
- Input Protection
- Bounds Checking
- Obfuscation

5.1.1 Address Protection

The address protection schemes share the assumption that addresses (e.g. return address) are critical data and must be tagged. In these schemes the metadata is created by functions that create the address (e.g. call instruction), and verified by the many instructions that use the address (e.g. return instruction). The schemes within this group are differentiated by the types of metadata they use (e.g. software managed and hardware supported. Methods with hardware supported metadata are StackGuard [6], ProPolice [8], StackGhost [10], RAS [18], [43], [44], SmashGuard [45], SCACHE [14], Split Stack [43], and Hardware Supported PointGuard [26],.

5.1.2 Input Protection

The input protection schemes are all in hardware and are the most promising. These schemes assume that external data are untrustworthy and should not be used as internal control data. The underlining concept is that "All input is evil until proven otherwise" [13]. In most cases, metadata are tightly coupled to the data in hardware (e.g. tagged memory). Data from external sources are tagged so it can be recognized, if there is an attempt to use it as control data. The schemes in this group differ in the management of metadata.

The underlying assumption is that that input data should be treated differently from local data, and should not be used as control data. Four methods Minos [4], [5], Tainted Pointer [2], Dynamic Flow Tracking [28], Dynamic Tainted Analysis [19], and Secure Bit [22] share the same assumption, but different implementations. 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. Finally, Secure Bit treats data passing between processes through the kernel as input.

Tainted Pointer additionally tried to prevent input from being used as a pointer. However, input is sometimes used as a part of pointer arithmetic (e.g. indexing) so protecting pointers can lead this scheme to break many programs.

Secure Bit protects a process from external control data, but does not prevent buffer-overflow attacks on non-control data. That raises the question: can an attacker use a buffer-overflow attack on non-control data to manipulate local control data to modify control flow?

5.1.3 Bounds Checking

Rather than tagging data, bounds checking schemes explicitly bound buffers to prevent overflow. In this case, the metadata is associated with every block of allocated data and is used to bound accesses. A classic hardware approach uses segmentation, but the granularity of the hardware usually isn't sufficiently fine grained (e.g. a function pointer within a data structure)—a notable exception is the i432 [30]. Array Bounds Checking [15] can be done entirely (and with some cost) in software or can be done with hardware support. Type-safe programming languages are examples of entirely software approaches.

5.1.4 Obfuscation

Instead of protecting the data directly, obfuscation schemes reorganize memory to obscure memory, making malicious manipulation of memory through buffer overflows more difficult. These schemes assume that attackers rely on a certain snapshot of addresses to overflow the critical data. If the snapshot is random or difficult to guess, an attack is more difficult. Being a software approach, it is easy to deploy. However, this scheme, unlike some hardware approaches, is vulnerable to brute force. Address Obfuscation [1] and ASLR [20] are good examples.

6. HARDWARE PROTECTION

Up to this point, we have covered the wide variety of protection schemes and the range of buffer-overflow attacks. In this section, we will individually analyze each hardware protection scheme. Based on our taxonomy, contemporary hardware solutions lie in four categories: isolation, address protection, bounds checking and input protection. We will briefly show the weakness of each one.

6.1 Isolation

Isolation schemes isolate the attacker either to eliminate an attack vector or to contain damage after a successful attack. Preventing the execution of code in stack memory isolates the stack from the attacker. Alternatively, limiting the memory of a process can isolate a compromised process. NX non-executable memory is an example of the former while sandboxing is an example of the latter.

Neither multistage buffer-overflow attacks nor arbitrary copy attacks require injection of code so they will work in the face of isolation schemes. In fact, many simpler attacks work.

Non-Executable Memory

Many non-X86 processors such as SPARC support non-executable memory, and AMD has recently added a similar feature named "NX" [17]. Non-executable memory prevents code in the buffer on the stack from being executed, effectively protecting against a class of buffer overflow attacks that executes code in the buffer on the stack. However, the integrity of the return address is not protected—leaving the system vulnerable to attacks using the address of either a resident shell or code in the heap. In certain cases, such as a signal handler return on Linux, the system requires an executable stack in order to function properly. Moreover, any LISP-like functional language requires an executable stack in their normal operation (a.k.a. trampoline). As a result, this method only protects against a narrow range of attacks.

Secure Code Installation

Instead of protecting the data, a Secure Program Execution Framework [16] (SPEF) aims at making a system difficult to inject malicious code. SPEF is a platform that consists of hardware mechanisms and compilation tools. The installation of a program requires both encryption and transformation. As a result, injecting the malicious code is not simple and requires a special process. This method prevents the injection of malicious code. Nonetheless, we have shown that it is possible to overflow the buffer and modify the return address or the function pointer to

point to a known address without injecting any code. Similar methods include [32] and instruction-set randomization [33]

Sandboxing

Sandboxing is a policy-enforcement mechanism. Since buffer-overflow occurs when information is passed from one domain to another domain, sandboxing a process intuitively cannot prevent such attacks. With appropriate policy rules, it is, however, possible to limit the damage of buffer-overflow attacks. Sandboxing can be done at several levels: kernel level [34], user level [35], or even hardware-supported sandboxing (e.g. Intel LaGrande [36], TCPA [37], [38], TrustZone [38], Microsoft NGSCB [39], ChipLock [40], Bear [41].) Like tagged memory, there exists a very fine-grained approach to memory management (e.g. MMP [42]), but such approaches can be successful for buffer-overflow protection only if a perfect combination of a security policy and an implementation exists. We believe that it is complementary to other techniques rather than a replacement.

6.2 Hardware Enhanced Address Protection

There are three types of hardware enhanced address protection schemes: Address Encoding, Copy of Address and Tags.

PointGuard [7] and Hardware Supported PointGuard [26] are good example of Address Encoding. They use a pre-defined key to encode every pointer before storing in memory and decode it before dereferencing. Ignoring the performance and compatibility issue, it is still possible to modify data using pointer arithmetic without overflowing the pointer (e.g. overflowing a variable i would allow us to create an arbitrary pointer if there is code such as "*ptr = ptr+i"*). Hence, a naïve arbitrary copy should be able to bypass this type of protection.

StackGhost [10], RAS [18], [43], [44], SmashGuard [45], SCACHE [14], and Split Stack [43] use a hardware redundant copy of a return address (such as cache memory, register window, return address stack, or hardware stack) for validating the return address. Ignoring the dynamic update of return address (e.g. non-LIFO control flow [21]), this mechanism only prevents the simple stack smashing attack. In fact, there exists an attack that modifies the exception pointer and totally bypasses the protection mechanism by bypassing the handling routine.

Here, we will present the concepts embedded in the IBM System/38 [46] as a representative of protection against buffer-overflow attacks provided by tagged architectures. In general, a hardware bit is used to indicate a type of data. On creation of a return address a call instruction sets the tag of a return address. Similarly with a function pointer, a special instruction sets the tag of a function pointer. Control instructions validate the tag bit before using it as a control address. Though return addresses and function pointers cannot be overflowed directly, an arbitrary copy is sufficient for bypassing this mechanism since no function pointer is being created—only an existing pointer is used.

6.3 Bounds Checking

Limited hardware protection has existed in various processors for many years, e.g. segmentation. Segmentation is primarily used as a mechanism to support the relocation of memory. In the early implementation of segmentation, a base register was required for each memory access. IA-32 and I432 [30] also adopted the idea and associated segmentation with base address, boundary check, and rings. By explicitly declaring and associating every buffer with a base and boundary, segmentation can protect against buffer-overflow attacks. A drawback of segmentation is the extra storage for storing segment descriptors. In IA-32, every memory access (in protected mode) requires a base and limit. However, most operating systems (e.g. Windows and Linux) bypass segmentation by setting one large segment for all memory in order to maintain portability and gain better performance. I432 was a CISC architecture that was designed with security awareness. Based on the paradigm of the ADA programming language, it checked every data boundary and forced every function call to create a new domain (segment). Since I432 instructions are bit encoded, ranging from six to 321 bits, computation took 10 to 20 times as long as the contemporary VAX 11/780 [47]. Consequently, I432 was a commercial failure. A similar concept can also apply to a function pointer. For example, one of the 1960s architecture, ICL 2900 series systems [48], had a native hardware 'pointer' type (a.k.a. descriptor) that included in it the size of the object pointed to. The hardware would check that any dereferences were not out of bounds. Bounds checking could prevent both attacks we present, but the commercial market has decided that a bounds check of *every* reference carries too much overhead.

6.4 Input Protection

Some methods assume that input data should be treated differently from local data. As a result, a bit is used for tracking the data passing across domains and preventing it from being used as control data. Examples include Secure Bit [22], Minos [4], [5], Tainted Pointer [2], and Dynamic Flow Tracking [28]. Since the malicious nature of buffer overflow attacks includes the modification of control data using input, this concept is able to protect against a large class of buffer-overflow attacks. A reasonable argument can be made that this type of approach currently provides the best protection against buffer-overflow attacks. For example, the address in the table of the multistage attack can be protected using input protection (but not all the implementations achieve that). However, arbitrary copy escapes handling because no input is being used as control data.

7. POINTER PROTECTION

Arbitrary Copy shows that viable attacks can occur by manipulating pointers which are not used (directly) for control so it is critical that pointers must also be protected from overflow. Unlike control data, pointers can be arbitrarily derived from input (a good example is array indexing). While input protection is shown to be useful for protecting buffer-overflow attacks on control data, we cannot apply the same concept directly to protect pointers. Differentiating between "good" and "bad" use of pointers is not possible without input from the user or compiler.

8. CONCLUSION

Hardware buffer-overflow protection provides greater protection than existing software schemes. In addition, the hardware mechanisms themselves are more difficult to attack. However, holes in the defense remain. In this paper we presented two attacks which even the new hardware schemes cannot prevent (some prevent one, but most prevent neither).

9. REFERENCES

[1] BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. 2003. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proc. of the 12th USENIX Security Symposium*.

[2] CHEN, S., XU, J., NAKKA, N., KALBARCZYK, Z., IYER, R. K. 2005. Defeating Memory Corruption Attacks via Pointer Taintedness Detection, in *Proc. Of IEEE International Conf. on Dependable Systems and Networks (DSN)*, Yokohama, Japan, June 28 - July 1, 2005

[3] CHIEN, E. AND SZÖR, P. 2002. Blended Attacks Exploits, Vulnerabilities and Buffer-Overflow Techniques in Computer Viruses. In *Proc. of Virus Bulletin Conf.*

[4] CRANDALL, J.R. AND CHONG. F.T. 2004. Minos: Control Data Attack Prevention Orthogonal to Memory Model. *Intl. Sym. on Microarchitecture*.

[5] CRANDALL, J.R. AND CHONG. F.T. 2005. A Security Assessment of the Minos Architecture. *ACM SIGARCH, Vol 33. No. 1*

[6] COWAN, C., BEATTIE, S., DAY, R. F., PU, C., WAGLE, P., AND WALTHINSEN, E. 1999. Protecting Systems from Stack Smashing Attacks with StackGuard. the Linux Expo, Raleigh, NC

[7] COWAN, C., BEATTIE, S., JOHANSEN J., AND WAGLE, P. 2003. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proc. of the 12th USENIX Security Symposium*.

[8] ETOH, J. 2000. GCC extension for protecting applications from stack-smashing attacks. IBM

[9] Flawfinder, http://www.dwheeler.com/flawfinder/

[10] FRANTZEN, M.S. M. 2000. StackGhost: Hardware facilitated stack protection. *In Proc. of the 10th USENIX Security Symposium*

[11] Glew, A. 2003. "Segments, Capabilities, and Buffer Overrun Attacks," Computer Architecture NEWS, *ACM SIG Computer Architecture Vol.31, No.4* - September 2003, pp. 26 – 31

[12] HAUGH, E. BISHOP, M. Testing C Programs for Buffer Overflow Vulnerabilities. In *Proc. of the 2003 Symposium on Networked and Distributed System Security (SNDSS 2003)* (Feb. 2003)

[13] HOWARD, M. AND LEBLANC, D. 1965. Chapter 10:All Input Is Evil!. Writing Secure Code, Microsoft Press, 2nd ed.(1965)

[14] INOUE, K. 2005. Energy-Security Tradeoff in a Secure Cache Architecture Against Buffer Overflow Attacks. *ACM SIGARCH, Vol 33. No. 1*

[15] JONES, R. W. M. AND KELLY, P.H.J. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In The *3rd Intl. Workshop on Automated Debugging*.

[16] KIROVSKI, D. DRINIC, M. AND POTKONJAK, M. 2002. Enabling Trusted Software Integrity. ACM *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*

[17] KRAZIT, T. 2004. PCWorld - News - AMD Chips Guard Against Trojan Horses. IDG News Service.

[18] MCGREGOR, J. P., KARIG, D. K., SHI, Z., AND LEE, R. B. 2003. A Processor Architecture Defense against Buffer Overflow Attacks. In *Proc. of the IEEE Intl. Conf. on Information Tech.: Research and Education (ITRE 2003)*, 243-250.

[19] NEWSOME, J., AND SONG, D. 2005. Dynamic Taint Analysis: Automatic Detection and Generation of Software Exploit Attacks. In *NDSS* (Feb, 2005)

[20] PAX TEAM. 2003. Documentation for the PaX project

[21] PIROMSOPA, K. AND ENBODY, R. 2004. Buffer Overflow: Fundamental. Technical Reports #MSU-SE-04-47, Department of Computer Science and Engineering, Michigan State University

[22] PIROMSOPA, K. AND ENBODY, R. 2005. Secure Bit2 : Transparent, Hardware Buffer-Overflow Protection. Technical Reports #MSU-CSE-05-9, Department of Computer Science and Engineering, Michigan State University (2005)

[23] PIROMSOPA, K. AND ENBODY, R. 2006. Survey of Buffer-Overflow Protection Technical Reports #MSU-CSE-06-3, , Department of Computer Science and Engineering, Michigan State University (2006)

[24] SCHMIDT, C., AND Darby, T. The What, Why, and How of the 1988 Internet Worm, http://www.snowplow.org/tom/worm/worm.html

[25] RATS, http://www.securesw.com/rats/

[26] SHAO, Z., ZHUGE, Q., HE, Y., SHA, E. H.-M. 2004. Defending Embedded Systems Against Buffer Overflow via Hardware/Software. In *Proc. of the 20th Annual Computer Security Applications Conference*, Tucson, Arizona (Dec. 6-10, 2004)

[27] SOLAR DESIGNER. 2002. Linux kernel patch from the Openwall Project (Non-Executable User Stack). http://www.openwall.com/

[28] SUH, G., LEE, J., AND DEVADAS, S. 2004. Secure program execution via dynamic information flow tracking. In *ASPLOS XI* (Oct, 2004.)

[29] VIEGA, J. BLOCH, J.T. KOHNO, Y AND MCGRAW, G. 2000. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Proc. of the 16th Annual Computer Security Applications Conferenc*e.

[30] ORGANICK, E. 1983. A programmer's View of the Intel 432 System, McGraw-Hill

[31] HSIANGREN, S. 2002. Apache/mod_ssl (slapper) Worm. GIAC Certified Incident Handler.

[32] MILENKOVIE, M., MILENKOVIC, A., JOVANOV, E. 2005. Using Instruction Block Signatures to Counter Code Injection Attacks. *ACM SIGARCH*, Vol 33. No. 1

[33] KC, G. S., KEROMYTIS, A. D. AND PREVELAKIS, V. 2003. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proc. of the 10th ACM Conf. on Comp. and Comm. Security*

[34] PETERSON, D. S. BISHOP, M. AND PANDEY, R. 2002. Flexible Containment Mechanism for Executing Untrusted Code. In *Proc. of the 11th USENIX UNIX Security Symposium*

[35] CHANG, F. ITZKOVITZ, A. AND KARAMCHETI, V. 2000. User-level Resource-constrained Sandboxing. *USENIX Windows System Symposium*

[36] Intel Corporation. 2003. LaGrande Tech. Architectural Overview.

[37] MACDONALD, R., SMITH, S. W., MARCHESINI, J. AND WILD, O. 2003. Bear: An Open-Source Virtual Secure Coprocessor based on TCPA. *Tech. Report TR2003-471*, Department of Computer Science, Dartmouth College.

[38] Trusted Computing Platform Alliance. 2004. TCPA IT White paper.

[39] Microsoft Corporation. 2004. The Next-Generation Secure Computing Base: An Overview.

[40] KGIL, T., FALK, L., MUDGE, T. 2005 ChipLock: Support for Secure Microarchitectures. *ACM SIGARCH*, Vol 33. No. 1

[41] MACDONALD, R., SMITH, S. W., MARCHESINI, J. AND WILD, O. 2003. Bear: An Open-Source Virtual Secure Coprocessor based on TCPA. *Tech. Report TR2003-471*, Department of Computer Science, Dartmouth College.

[42] WITCHEL, E., CATES, J. AND ASANOVIC, K. 2002. Mondrian memory protection. In *ASPLOS-X*, Oct 2002.

[43] XU, J., KALBARCZYK, Z., PATEL, S., AND IYER, R. K. 2002. Architecture Support for Defending Against Buffer Overflow Attacks. In *Workshop on Evaluating and Architecting Systems for Dependability.*

[44] YE, D., KAELI, D. 2005. A Reliable Return Address Stack: Microarchitectural Features to Defeat Stack Smashing. *ACM SIGARCH*, Vol 33. No. 1

[45] OZDOGANOGLU, H., VIJAYKUMAR, T.N., BRODLEY, C.E., JALOTE, A. AND KUPERMAN, B. A. 2003. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. Tech Report (TR-ECE 03-13), Department of Electrical and Computer Engineering, Purdue University.

[46] DAHLBY, S.H. HENRY, G.G. REYNOLDS, D.N. AND TAYLOR, P.T. 1982. Chapter 32. The IBM System/38: A High-Level Machine. Computer Structures: Principles and Examples.

[47] COLWELL, R. P., ET AL. 1985. Instruction Sets and Beyond: Computers, Complexity and Controversy. *IEEE Computer*.

[48] GENHRINGER, E. F. AND KEEDY, J. L. 1985 Tagged architecture: how compelling are its advantages?. *Intl. symposium on Computer architecture*, pp. 162-170