Secure Bit2[1]: Transparent, Hardware Buffer-Overflow Protection

"Give me a (little) bit, and I will solve buffer overflow."

Krerk Piromsopa and Richard J. Enbody

Department of Computer Science and Engineering

Michigan State University

East Lansing, Michigan 48824, USA

{piromsop, enbody}@cse.msu.edu

Abstract— We propose a new, minimalist, architectural approach, *SecureBit2*, to protect against buffer overflow and function-pointer attacks. Secure Bit is a concept to provide a hardware bit to protect the integrity of addresses for the purpose of preventing buffer-overflow attacks. SecureBit2 is our second implementation of a protocol to manage the Secure Bit. SecureBit2 is completely transparent to software, and provides 100% backward compatible with legacy code. Unlike several methods that only reduce the probability of a successful attack, SecureBit2 can detect and prevent all buffer-overflow attacks. SecureBit2 is transparent to software, and has little run-time performance penalty (almost none). The goal of SecureBit2 is to provide hardware support to protect against current and future generations of buffer-overflow attacks by protecting the integrity of addresses: addresses passed in buffers between processes are invalid. Included is our proof that validates the mechanism of the SecureBit2. Robustness and transparency are demonstrated by emulating the hardware, and booting Linux on the emulator, running application software on that Linux, and performing known attacks.

---

1 Patent Pending

Keyword— Buffer overflow, Buffer-Overflow Attacks, Function-Pointer Attacks, Intrusion Detection, Intrusion Prevention

1 Introduction

We propose a new, minimalist, architectural approach, *secure bit 2*, to protect against buffer-overflow attacks (return-address attacks and function-pointer attacks). It is a continuation of our original work on Secure Bit [69, 70]: both are based on an added Secure Bit, but the management of the bit is dramatically different. We refer to the new *management* scheme as Secure Bit 2. Secure Bit is completely transparent to software, and has no (or relatively small) run-time performance penalty. The goal of Secure Bit is to provide hardware support to protect against current and future generations of buffer-overflow attacks by protecting the integrity of addresses. In this paper we present a scheme to manage Secure Bit. Included is our proof that validates the mechanism of the Secure Bit. Robustness and transparency are demonstrated by emulating the hardware, and booting Linux on the emulator.

Since the 1988 exploitation of buffer overflow in fingerd by the Morris Worm [1], buffer overflow has persisted as the basis for many major attacks and new variations continue to emerge [2, 4, 5]. Puncus and Baker [2] include an interesting analysis of buffer-overflow attacks. Examples of buffer-overflow attacks in the real world include an SSL exploit [7], Apache Slapper [8], Security Software [9], JPEG Processing (GDI+) [10], to name just a few.

Prevention requires either perfect software or some meta-information which can reside in software or hardware [3]. Currently, there are many software and hardware solutions to the problem [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 37, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56]. A software solution such as the

popular StackGuard [11, 12, 13, 14] is a good choice for first and second generation buffer overflows [5] when no appropriate hardware implementation is available. However, the current solutions mostly miss the basis of buffer-overflow attacks and only indirectly address the problem. Thus, they provide only partial protection. Ignoring the performance issue, here are examples:

- Those methods that prevent injecting of malicious code [46, 47] (including non-executable memory area [15, 16, 17, 18] exemplified recently by NX and memory reorganization [17, 19, 20]) cannot prevent the ARC injection (where the malicious code is resident code or shared libraries.) [2, 21]. Also, they do not protect against overflow attacks on function pointers.
- Those methods that put a guard value before the return address [11, 12, 13, 14] exemplified by StackGuard can only protect against stack smashing, but not the overflow that skips the guard value (e.g. addressing a structure/word boundary data). Also, they do not protect against overflow attacks on function pointers.
- Those methods that use a compiler-generated key or a per-process key [11, 12, 13, 14, 22, 23, 24], cannot support shared libraries and have trouble protecting the key (also trouble protecting the key-generator algorithm). Moreover, a combination of information-leaking attacks (also referred as Read Attacks [24]), brute-force attacks, and replay attacks may allow us to obtain the key (table) and reverse engineer the encoding algorithm, and successfully compromise the system. (See section 2.3 for more details.)
- Those methods that validate the return address with a redundant copy [25, 26, 27, 28, 29, 30, 31, 32] as well as separating the stack [26] are not compatible with trampolines (including functional languages such as LISP), non-LIFO flow control such as is found in the Linux kernel (which does not use call/return pairs) and some

dynamic memory management. Also, they do not protect against overflow attacks on function pointers.

- Those that control memory access boundaries [33, 34, 35] may have performance issues if done entirely in software, and complicated management of boundaries especially with complex structures and their nested boundaries. Similarly, hardware enforced boundaries (e.g. segmentation) [23, 36, 37] can result in many small segments, have trouble with nested structures, and, in some cases, can restrict the programming model. Also, they may not protect against overflow attacks on function pointers.
- Those that use abstract data types or are interpreted on a virtual machine (including type-safe programming languages) do not solve the problem, but simply raise the problem up a level. Since the system eventually has to interface with the low-level libraries, operating system, and hardware, the type-safe programming languages are not bullet proof with respect to buffer-overflow attacks (e.g. PERL [38], Java [39, 40], dotNet [10]). Also, they do not protect against overflow attacks on function pointers.
- Sandboxing methods (including hardware supported sandboxing, e.g. Intel LaGrande [41] and MS NGSCB [42], TCPA [43, 44], TrustZone [45]) do not prevent buffer-overflow attacks. The mechanisms only limit the damage that may be caused by the attacks.
- Static Analysis methods [48, 49, 50, 51, 52, 53, 54, 55, 56] can only protect against a set of known vulnerabilities, but not the next generation. Also, they may not protect against overflow attacks on function pointers.

Moreover, they all are mostly incompatible with non-LIFO control flow (e.g. longjmp, and signal handling). We conclude that any single existing approach or combination of

approaches fails to protect against a full range of buffer-overflow attacks. Thus, a hardware solution that provides (better) protection with a reduced performance penalty is needed.

Carefully analysis of all types of buffer-overflow attacks indicates that the vulnerability is related to the preservation of the integrity of addresses (return address and function-pointer addresses in particular). To be specific, the attacks always modify either a return address or a function pointer by using a buffer passed from another domain (machine, process). Note that preventing the injection of code is not a necessary condition for a buffer overflow attack [2]. Therefore, the underlying concept of buffer-overflow protection can be summed up as: *to protect against buffer-overflow attacks is to preserve the integrity of addresses.*

Unlike several existing mechanisms, the Secure Bit provides protection mechanism and supports non-LIFO control flow without any modification to legacy code and libraries. ([69] has a good analysis of non-LIFO control flow) Only a trivial modification is necessary to enforce the protection mechanism: a modification to a few small routines in the kernel.

We claim that Secure Bit can prevent current and future generations of buffer-overflow attacks, and support that claim with a formalization of the concept and a proof. An important characteristic is that Secure Bit is completely transparent to software, 100% backward compatible, and can prevent buffer-overflow attacks (not just reduce the probability of a successful attack.) The obvious disadvantage is that the price of this security and compatibility is a change in hardware. However, these changes are small in comparison to other hardware approaches such as Intel's LaGrande. (Note that Secure Bit can enhance LaGrande.)

This paper is organized as follows. Section two provides background on buffer-overflow attacks and a literature review. Sections three and four detail Secure Bit and related issues. Section five is the implementation. Sections six and seven are the evaluation and the analysis. In the last section, we conclude that Secure Bit can prevent the system from heterogeneous generations of buffer-overflow.

2 Background

This section is intended to be a gentle introduction to buffer overflows. We begin by exploring buffer overflows in general, and return-address attacks and function-pointer attacks in particular. Later in this section, we review several approaches currently used by programmers and architects to defeat buffer overflow by generally explaining a concept and pointing out the strengths and weaknesses of each approach. In the end, we conclude that buffer overflow still needs a better solution.

2.1 Buffer-Overflow

A definition of buffer overflow is presented in Definition 1 (from the Webopedia Computer Dictionary [67]).

Definition 1:

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 that is already contained in that buffer.

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

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.

Initially, the attacked address was a return address, but later function pointers were attacked. In either case, the eventual access of that address (e.g. by a return or 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 or a Trojan horse working within the kernel.

2.2 Prevention

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

Postulate 1:

In buffer overflow attacks, the generic buffer/memory-manipulating operations are used by the vulnerable routine to overflow the address (e.g. a return address and 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 and 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. Handling the signature on a host system is the focus of the rest of the paper.

Specifically, if local data and data passed from another domain can be differentiated, we can detect buffer overflow. 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.

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

Proof:

Theorem 1 follows directly from Definition 1, and Definition 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.

2.3 Related Works

At this point, we have established definitions of the attacks and will continue to explore current approaches against buffer-overflow attacks. To elucidate the wide variety of approaches, the methods are categorized to form a taxonomy (in Figure 1). For each class in the taxonomy, we briefly discuss the mechanisms and the (potential) problems.

In the big picture, there are two types of solutions against buffer-overflow attacks [57, 58]: static analysis, and dynamic detection/prevention. Static analysis refers to methods that use static knowledge of known attacks, and try to prevent the potential vulnerability before a program is deployed. Dynamic detection/prevention refers to methods that use the knowledge

of the run-time environment to either detect or prevent the problem. Figure 1 shows the classification tree.

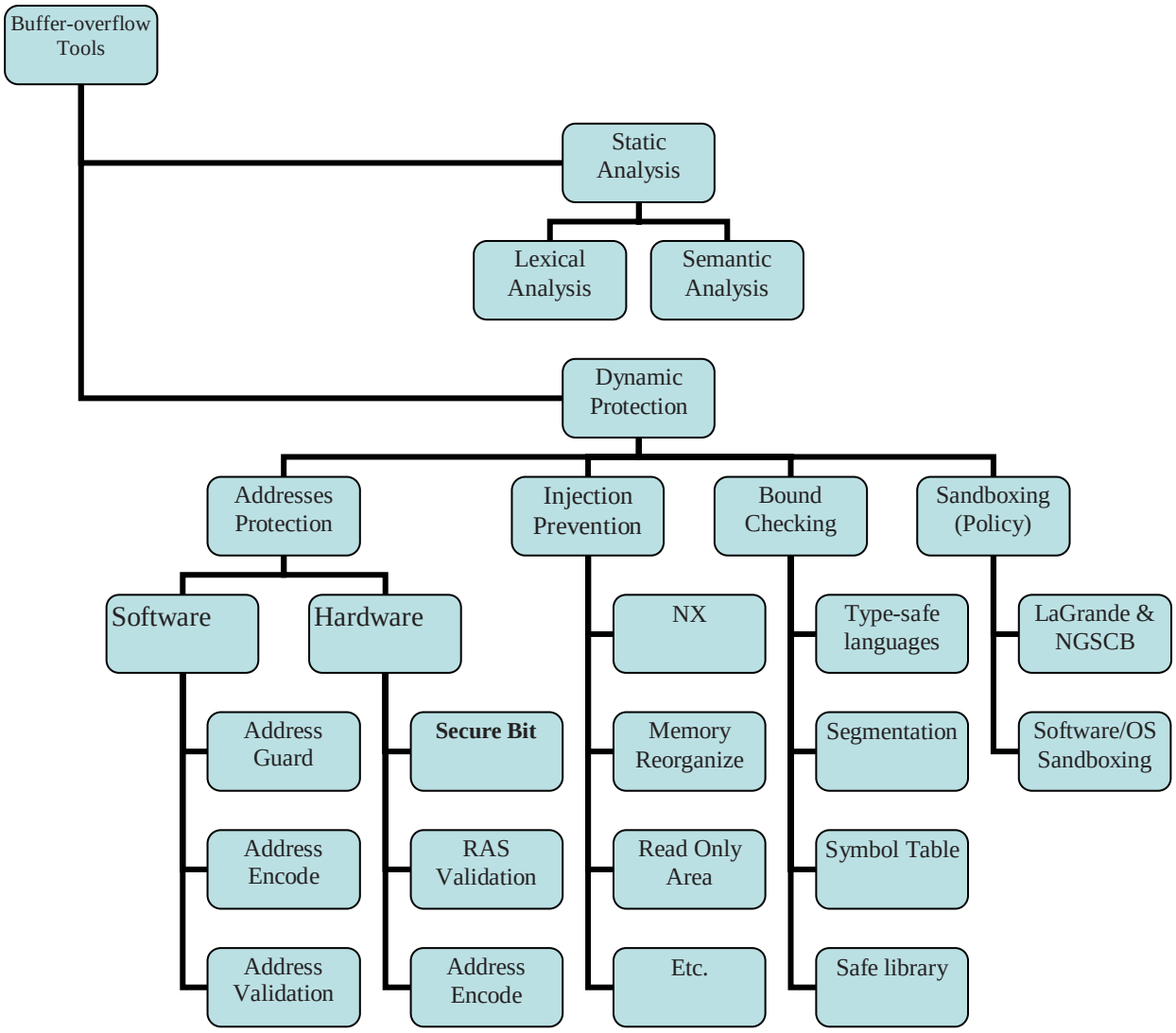

Figure 1 Taxonomy of solutions against buffer-overflow attacks

Static Analysis tools [49, 50, 51, 52, 53, 54, 55, 56] generally examine source code to detect possible buffer-overflow vulnerabilities. Basically, static analysis scans and analyzes at the source level with predefined profiles. The analyzed results provide a report to programmers for modifying the program. The method can be a simple pattern-matching algorithm [49], lexical analyzer [50, 51], or scanner (parser) [52]. The parser tools tend to have a better chance of differentiating between correct and incorrect use of functions than those based on lexical analysis and pattern matching. A benefit of Static Analysis is that it allows a

programmer to prevent the problem before deploying the program. However, the static analysis has no runtime information. As a result, it might not be able to evaluate all possible problems and may generate false alarms. No matter how good the static analysis tool is, a programmer, eventually, has to make a final decision in correcting the program logic.

Using approaches as criteria, dynamic tools can further be classified into four schemes: Addresses Protection, Injection Prevention, Bound Checking, and Sandboxing.

Address Protection schemes fundamentally detect if the address is maliciously modified. Such detections can result from placing a guard value adjacent to the address [11, 12, 13, 14, 20], encoding the address [22, 23, 24], or validating it with a redundant copy (which is either created by the software itself [25, 29, 30, 31, 32] or using the feature of the processors such as Return Address Stack [26, 27]). By placing a guard value adjacent to the address, attackers can still damage the address without touching the guard value [59] or even reproduce a valid guard value themselves [60]. In addition, the encoding method also has trouble with protecting the encoding key and algorithm [59, 60]. Ignoring the compatibility and shared libraries issues, a compiler-generated key/guard value is usually a fixed value, which can be easily determined by disassembling the program. Similarly, a per-process key/guard value still suffers with key management and function pointers [59, 61], examples of how to attack the compiler generated key and per-process key can be found in [59, 60] and [59, 61] respectively. By validating the address with a redundant copy [25, 26, 27, 29, 30, 31, 32], a critical task is to manage the validating copy while supporting the dynamic modification of the address (which is normally used in non-LIFO control flow).

It is worth taking the time to comment in more detail on one scheme, PointGuard and its recently proposed hardware implementation [24]. With hardware support encoding and hardware protected per-process key (table), [24] is able to encode every function pointer (and return address) via binary modification. However, with a combination of information-leaking attacks (read attacks), brute-force attacks, and replay attacks, it is possible to obtain a key or reuse an encrypted data. In fact, several daemons (processes) are multi-threaded programs and have to use the same keys. Furthermore, the daemons are sometimes implemented to be fault tolerant by catching segmentation-fault signals – good coding which counter intuitively can aid an attacker. Thus, PointGuard (either hardware or software) can only reduce the probability of a successful attack.

Injection Prevention schemes either prevent the malicious code from being injected or prevent injected code from being executed. The underlying mechanisms can be hardware-supported, non-executable regions such as NX [15], or OS (and hardware) supported read-only areas (e.g. Solaris, BSD, Linux [17, 18]). Alternatively, the mechanism can be simply a random reorganization of memory [17, 19], Instruction encoding [46, 47], which makes the injection process more difficult. Nonetheless, all methods in this scheme cannot prevent ARC injection [2, 21]. In addition, they usually prevent trampolines from their normal execution.

Trying to prevent the buffer from being overflowed, Bound Checking schemes perform boundary checks on every memory access. Conceptually, a base pointer is defined for every pointer. A pointer value is valid for only one memory region. Checking whether the reference is in the same region as the one referred by a base pointer would enforce bounds checking. This method is useful to solve buffer-overflow attacks in general. With the overhead of a symbol table used to keep track of each pointer, it may experience more than 30 times

slowdown in a pointer-intensive program [33]. As a result, this tool is ideal for debugging, but may not be suitable for deployed applications. Alternatively, the solution can be simply a safe string-manipulation library [29] which dynamically calculates the length of each buffer. With a segment descriptor acting as a symbol table, segmentation [36, 37] is a hardware implementation of bound checking. Ignoring the performance issue, bound checking does not always prevent buffer-overflow attacks, since a buffer in a different context has different semantics. For example, manipulating a data structure with function pointers still allows ARC-injection. Some mechanisms which limit memory access to the current stack frame [23] can restrict the programming model. For the type-safe programming languages (e.g. Java, PERL, .Net), the checking process is usually embedded into the virtual machine. Usually, the virtual machine itself is written in C/C++ so somehow the virtual machine has to interact with standard libraries or components underlying type-unsafe languages. Type-safe languages can still be exploited by buffer overflows, but they do decrease the probability of being attacked. For example, we still experience buffer-overflow attacks in Java [39, 40], Perl [38], .Net [10], etc.

Sandboxing is a policy-enforcement mechanism. Since buffer-overflow occurs when information is passed from one function to another function, 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 [63], user level [62, 63], or even hardware-supported sandboxing [41, 42, 43, 44, 45].

We can conclude that current solutions provide partial protection or reduce the probability of a successful attack.

2.4 Sample Attacks

In this section, we will illustrate a type of buffer-overflow attack that can bypass most software solutions. Figure 2 is an example of multistage buffer-overflow attacks (similar to the SSL exploit founded in Apache Slapper [8]). Given a resident shell code, we will demonstrate an ARC-injection attack on the *printf* function by modifying the global offset table (Jump slot)—the table is used to dynamically bind the *printf* function to the code in a shared library (libc).

Algorithmically, the fundamental of multistage buffer-overflow is that there exists a vulnerable pointer to a buffer. First, the pointer is modified (by overflowing) to point to a specific location (e.g. a jump slot, a function pointer). In the second stage, an input is stored to the location pointed by this pointer. The first stage buffer overflow allows attackers to create a pointer to any location; the second stage will overwrite that location with any preferred value. Assuming that a program is compiled with shared library, modifying the jump slot will allow attacker to bind a function to any code. In case there is a software protection, modifying the jump slot of the handler routine also allows attackers to bypass the protection mechanism.

Figure 2 is an example of vulnerable program. To attack this type of program, the buffer-overflow is done in two stages. First, the ptr pointer is overflowed to point to any preferred memory location, e.g. shell code. Suppose, for the purpose of illustration, that we learn the address of resident shell code (*residentcode*). If we can modify a function pointer to point to this resident code, we can compromise a system.

```
#include <stdio.h>

int residentcode() {
       execl("/bin/sh","/bin/sh",0x00);
```

```
}

int vulnerable(char **argv) {
        int x;
        char *ptr;
        char buffer[30];
        ptr=buffer;
        printf("ptr %p - before\n",ptr);
        strcpy(ptr,argv[1]);    // overflow ptr with preferred address
        printf("ptr %p - after\n",ptr);
        strcpy(ptr,argv[2]);    // buffer overflow
}

int main (int argc,char *argv[]) {
        int i;
        printf("Sample program.\n");
        vulnerable(argv);
        printf("Program exits normally.\n");
}
```

Figure 2 Example of Vulnerable program

Figure 3 is an example of an exploit program. With some information obtained by using the *objdump* utility (for example), we can learn that the resident shell code (*residentcode*) and the *printf* jump slot are located at 0x08048454 and 0x08049730 respectively. Armed with that information, the attack program will first overflow the *ptr* pointer to point to the jump slot (in global offset table) of the *printf* entry, and then overflow the *printf* entry so it will contain the address of the resident shell code (*residentcode*). After the attack overwrites this jump slot, the next call to *printf* is redirected to the *residentcode*.

This kind of attack is similar to that found in Apache Slapper [8] where the Jump Slot of the *free* function was modified. The eventual execution of the *free* function results in a remote shell accessible from the attacker's machine.

```
int main(int argc,char **argv) {
        int *iptr;
        char *buf1 = (char *)malloc(sizeof(char)*46);
        char buf2[5]="Addr";
        char **arr = (char **)malloc(sizeof(char *)*4);

        memset(buf1,'x', 0x20);
        iptr=(int *) buf1;
        iptr+=(0x20 / sizeof(int));
        *iptr=0x08049730; // point to any function pointer
        buf1[0x24]='\0';
        iptr=(int *)buf2;
        *iptr=0x08048454; // address of resident code
```

```
        arr[0]="./vul";
        arr[1]=buf1;
        arr[2]=buf2;
        arr[3]='\0';
        execv(arr[0],arr);
}
```

Figure 3 Example of Exploit program

By creating a pointer to an arbitrary memory location, we can bypass most protection mechanisms. By modifying the offset table (Jump slot), we not only avoid those protections that only prevent stack smashing, but we can also bypass the handler routine in most software-protection schemes. This class of attack can also bypass the mechanism found in Microsoft Windows 2003 Server [61].

2.5 Summary

The buffer-overflow attack is a problem that was indirectly addressed in several ways. Among several variations, buffer-overflow attacks required overflowing addresses (return addresses and function pointers) with a buffer passed from another domain (machine, and process). As a result, ***a necessary condition for preventing buffer-overflow attacks is preservation of the integrity of addresses across domains***.

3 Fundamental of Secure Bit2

Fundamentally, we add one bit to every memory word to protect the integrity of addresses by adding semantic meaning to each word of the memory. This semantic meaning will be used to distinguish local data and data from another domain.

In this section, we first give a general concept of Secure Bit2. Later, we prove that the mechanism creates sufficient conditions for a secure system with respected to buffer-overflow attacks.

3.1 General Mechanisms

We add a Secure Bit to every memory location (and register). This bit is handled by the memory manipulating instructions as a regular memory word (moved along with the associated word). Except for those words in buffers passed between processes, such operations have to mark the Secure Bit at the destination (either a register, or a memory location). Words in buffers passed between processes get their secure bit set. Call, return, and jump instructions check the secure bit and if the secure bit is set, the processor issues an interrupt or fault signal. Figure 4 shows a memory snapshot with Secure Bit in various situations.

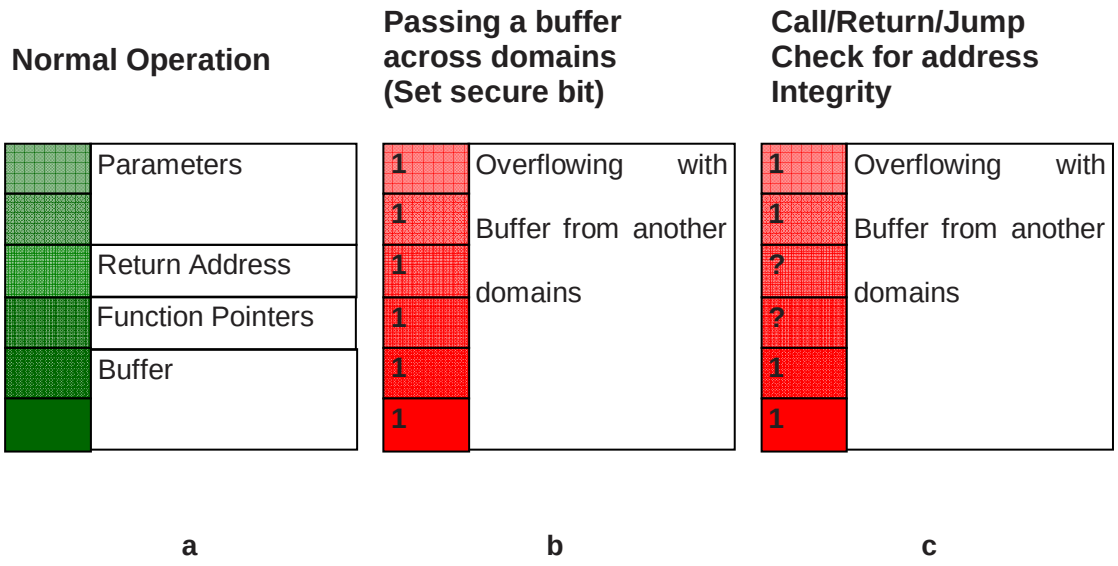

Figure 4 Memory Snap shot with Secure Bit (a) normal operation. (b) Passing a buffer across domains. (c) Related instructions validate the address

By setting the Secure Bit in a buffer passing across domains, associated instructions can easily detect that an address (a return address or a function pointer) was modified by a buffer passing from another domain—there is a buffer-overflow. For later reference, we establish the condition of setting the Secure Bit when manipulating data across domains as Protocol 1.

Protocol 1:

Passing a buffer across domains (devices, machines, and processes) always sets the Secure Bit.

We will prove that Protocol 1 is a sufficient condition for preventing buffer-overflow attacks in the next section.

To distinguish between normal memory manipulation (within the domain) and passing a buffer across domains, a mode of operation (*sbit_write*) is introduced to a processor. Manipulating memory will always set the Secure Bit when the *sbit_write* mode is set. Normal memory manipulating operation (when the *sbit_write* mode is clear) will carry the Secure Bit along with the associated memory words. With the presence of this mode, the kernel (or a process) can switch the mode of operation when handling the buffer from another domain. This scheme allows us to provide backward compatibility to all legacy code—mode changes are only necessary within an operating system so Secure Bit is transparent to all other code. We will elaborate the mechanism and necessary conditions for enforcing Protocol 1 in Section 3.3.

3.2 Proof of Concept

To claim that a system 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. – A definition from [64].

In the case of buffer-overflow attacks, the security policy is simply the statement: "Overflowing a buffer cannot create a valid address (e.g. a return address and 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. – A definition from [64]

Theorem 2:

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

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 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

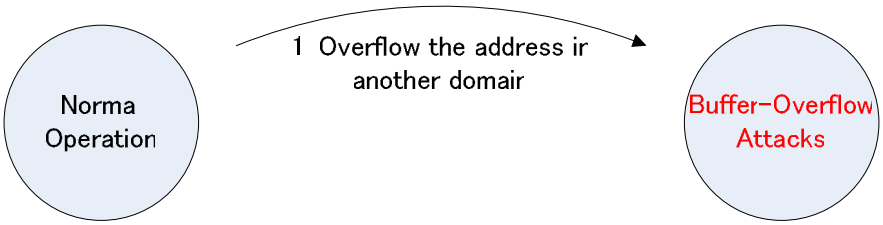

Figure 5 State-transition diagram of buffer-overflow attacks

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

Theorem 3:

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

Proof:

With the presence of the Secure Bit and 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. 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 Secure Bit preserves the integrity of the address and is a secure system with respect to buffer-overflow attacks. This follows directly from Theorem 2. QED

3.3 Protocol Enforcement

This section is the discussion of modifications necessary to enforce Protocol 1. We first elaborate the definition of passing data across domains in our context. From this definition, we provide a guideline for necessary modifications required to enforce such a protocol.

To ease understanding the definition of passing data across domains, we first introduce the term “threat surface” found in Threat Modeling [68]. By invoking the threat modeling into the life cycle of software engineering, “threat surface” is defined as all possible input crossing from the software interface. In this context, a domain is a boundary with respect to the current process, and passing data across domains means interfacing the software with other components – it is “threat surface”. Intuitively, passing data across domains not only includes passing a buffer between processes (regular IPC), but it also includes reading and writing from I/O devices, passing a command-line argument to a new process, sending and receiving data from a network socket, etc. Though it sounds somewhat complex, the fundamental concept is that every buffer that is used to interface between the software and outside components is data passing across domains and is therefore suspect.

With this fundamental concept in hand, we can easily analyze the threat surface of software (the O.S. in our case) and modify the surface to operate in *sbit_write* mode: set the Secure Bit to every memory word passing across domains. The set of access points between processes (the threat surface) is easily identified because they operate across different segments or pages (or any other future type of memory protection structure). In this case, the buffers on the threat surface always pass though the kernel. Thus, we can simply enforce Protocol 1 by modifying the instructions that move data across domains to operate in the *sbit_write* mode. (Section 6.1 summarizes such modifications)

4 Issues

In this section, we analyze issues related to the implementation and the mechanisms of the Secure Bit. These issues include Virtual Memory and Memory Interface.

4.1 Virtual Memory

With the presence of virtual memory, a page of memory can be swapped to the swap space. As a result, a secure bit has to be swapped in and out of main memory. Handling this situation can be done by introducing an instruction to check the status of the secure bit and allocate additional space for storing the secure bits. (A separate bitmap is likely an efficient way to store the bits—128 bytes for 4K bytes of memory.)

Note that the space for storing the secure bit is only required for the Stack and the Heap. The text section does not need any swap space for the secure bit. The paging routine will be modified to render the bitmap of the secure bit on swap out, and use the *sbit_write* mode to restore the secure bit on swap in.

Without a direct interface to the user, attacking the swapper with a buffer overflow attack is impossible. Similarly, modifying the swap space is protected (e.g. by the operating system, by the file permission). The latter protection is considered to be safe from buffer-overflow attacks. Another way of looking at it is to observe that these routines are far removed from the threat surface with buffer passed to them from outside.

Though execution time will be increased for swapping a process, the complexity is relatively small compared to other methods. Furthermore, a hardware/software optimization plays a

role here. Regardless of the size of a generic hard drive today, the additional space is not an issue.

4.2 Memory interface

With the presence of Secure Bit, this additional bit intuitively requires an additional pin added to the memory chip. To ensure the integrity of the Secure Bit, this pin is associated with operations that modify the Secure Bit. It functions as both input and output at the same time. Similarly, processors also require a pin to access this Secure Bit. The general system organization would look like the diagram in Figure 6 (a)

Processors will control the access to the Secure Bit line according to instructions and mode as described earlier. With a memory hierarchy (e.g. cache), a cache controller can manage and interleave the Secure Bits like managing normal data bits.

Another issue is that the Secure Bit tends to create a data bus with an odd number of lines. Moreover, there is no off-the-shelf memory controller or memory chip that currently supports Secure Bit. As a transitional approach (which might be the best approach anyway), we can interleave data storage and semantic storage (Secure Bit) using a separate interface. While data lines have a byte boundary, the semantic line has a bit boundary. This separation allows us to create a simple memory controller that will convert a bit interface of the Secure Bit to access legacy, byte-boundary memory chips. Figure 6 (b) shows block diagram of the alternate memory interfaces.

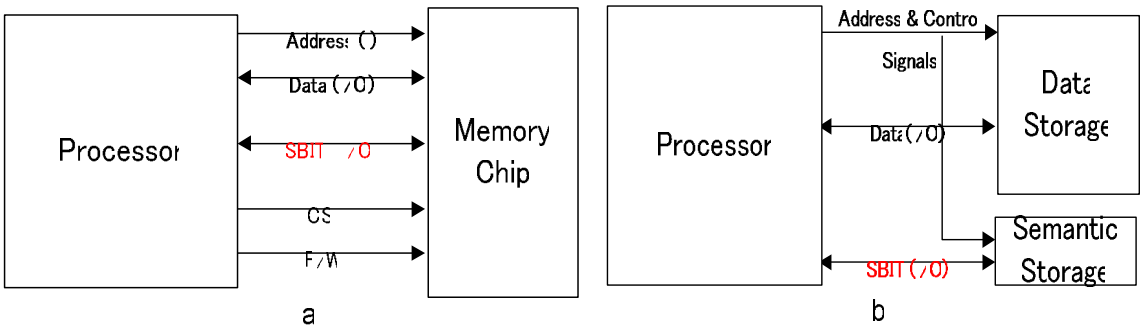

Figure 6 System block diagram (a) Interleaving Memory Interface (b)

4.3 Possible Attacks

For any proposed security scheme, one must always consider ways around it. With the presence of Secure Bit, buffer overflow can only occur using a buffer created by the process itself (e.g. assign a constant value to a pointer). Since Secure Bit is a part of each memory word and manipulating data passing from another domain always preserves the Secure Bit, using data passing from another domain (or its derived result) as a function pointer or a return address is prohibited. Since we have proved that an external attack is not possible, only an internal attack is a possibility (assigning local data or constant value). Such an internal attack must actively circumvent protections to maliciously attack itself. However, it is not an attack, but a "process suicide" (programming error). With respect to the definition of buffer-overflow attacks (definition 2) only using data passing from another is considered an attack.

5 Implementation

To demonstrate the completeness of Secure Bit, its compatibility, its transparency, and its effectiveness we implemented it in both an emulator and a hardware simulator. The best way to elaborate the details of Secure Bit is to create an implementation that is complete and that we can really attack. Implementing Secure Bit in an emulator provided a platform for demonstrating completeness, compatibility, and transparency because we could boot Linux on it. Moreover, we want to capture all possible hardware issues involved in investing in a

complex IC design so we also implemented Secure Bit in a hardware simulator. Together both the emulation and the simulation allow us to test Secure Bit with a real operating system under real attacks and to capture all hardware behavior.

5.1 BOCHS Emulator

Bochs [65] is an IA-32 emulator which is written in C++, is highly portable, and is easy to modify. Bochs provided a flexible platform for a sufficiently complete emulation that we could test real software (Linux) to test the completeness and correctness of our concept.

5.1.1 Instruction Set Architecture

To implement Secure Bit we needed to change the semantics of several instructions and add an instruction to the IA-32 Instruction Set Architecture to set the mode. With several combinations of addressing mode, boundary access, processors' mode and address translation, the full details are somewhat lengthy, but can be summarized as follows.

- The *sbit_write* flag is added to the EFLAGS register.
- The semantics of the *RETURN* instruction are modified to validate the Secure Bit of the return address and raise the protection flag when the Secure Bit is invalidated.
- The semantics of the *CALL* and *JUMP* instruction are modified to validate the Secure Bit of the address/register that holds the target address when the target is an indirect value (a function pointer), and raise the protection flag when the Secure Bit is invalidated
- Other instructions that access memory are modified to carry the Secure Bit along with the memory word when the *sbit_write* mode is cleared, and to set the Secure bit at the destination when the *sbit_write* mode is set.

Although our implementation is in IA-32 there is nothing about a RISC ISA which would prevent a similar implementation. In fact, our hardware simulation indirectly supports that assertion.

5.2 Simple Scalar

SimpleScalar [66] is a register-transfer-level hardware simulator used to model applications for performance evaluation. SimpleScalar is the accepted standard in the architecture community for investigating processor-level implementations. It was designed for both maximum flexibility and effective monitoring of performance. We use SimpleScalar to capture processors' behaviors when the Secure Bit is present. These behaviors include memory hierarchy (cache), out-of-order execution (pipeline), and the impact on the ISA of a RISC architecture.

Like the emulator, we add another memory structure to SimpleScalar and modify associated instructions (as well as cache, registers, etc.) to interface with this structure. For example, SimpleScalar has a structure "mem_t" which describes memory so we could simply add an entry to that structure for Secure Bit.

With the hardware simulator we can implement Secure Bit at the processor level and investigate the low-level details of the impact of Secure Bit. A number of address traces exist for SimpleScalar to validate that the modified simulator is working correctly. (We do not have a trace of an attack, and we feel that such a trace is unnecessary given what we can establish that ability with the emulator.) With simpleScalar we can cover the issues not answered in the emulator implementation. In this way, we improve our measure of the performance and costs of Secure Bit.

6 Evaluation

Our assumption is that Secure Bit will protect against buffer overflow without degrading performance. Though it requires more memory, adding more memory is a one-time cost compared to the indefinite cost of future buffer-overflow attacks. We evaluate our work with several aspects of hardware and software issues as follows.

- An ability to boot Linux on a Secure-Bit-enabled machine demonstrates the completeness of our implementation in its ability to handle software of the complexity of a real operating system.
- An ability to run a legacy application on a Secure-Bit-enabled machine demonstrates the transparency and compatibility of Secure Bit from an application point of view.
- An ability to prevent an unmodified, buffer-overflow-vulnerable program from being exploited validates the effectiveness in preventing buffer-overflow attacks.
- The instructions that have to be modified in order to support Secure Bit indirectly demonstrate the necessary semantics for preventing buffer overflow attacks in hardware.
- The results from SimpleScalar demonstrate the impact of Secure Bit on the architecture, cache, memory hierarchy, pipeline and processor.

6.1 Booting Linux

We have booted Linux on the emulator to demonstrate the transparency of Secure Bit as well as its compatibility. With Linux running, we mount existing buffer-overflow attacks, and demonstrate how Secure Bit will trap them.

Since our architecture does not modify the syntax of any instruction, we are able to boot an unmodified version of Linux RedHat 6.2 (as well as other operating systems such as FreeDos, NetBSD, and Windows XP) in the emulator.

To enforce the protection mechanism, we modify the part of Linux kernel that passes a buffer across domains (copying data between a process and a kernel or a common threat interface of every process). With this modification, we cover a variety of communication: a buffer of a network socket, command-line arguments, a buffer passing from a process to kernel and vice versa, and a buffer passing between processes. All of these routines can be found in one file "/arch/i386/lib/usercopy.c" (and some preprocessing macro in the file "/include/asm-i386/uaccess.h"). The modifications can be summed up by listing the small set of routines to operate in *sbit_write* mode:

- __generic_copy_to_user
- __generic_copy_from_user
- __strncpy_from_user
- strncpy_from_user

6.2 Mounting Attack

Testing well-known buffer-overflow attacks requires specific versions of application software running on specific versions of kernels. Therefore, testing a suite of attacks is unrealistic on an emulator. Instead, we have created a simple program that is vulnerable to buffer-overflow attacks, and tried to mount several attacks on it. Without any modification Secure Bit trapped and prevented them all. In addition to normal return-address attacks and function-pointer attacks, we also mount an advanced attack (multistage buffer-overflow attack) that overflows the Global Offset Table (which can bypass existing protections as described above in Section

2.4). In fact, we manage to compile/install and remotely mount the SLAPPER [8] worm attack to the vulnerable version of Apache mod_ssl in the emulator. (SLAPPER [8] is a type of multistage buffer-overflow attack, which modifies the Global Offset Table and can also bypass any protection mechanism.) All attempts failed to compromise the Secure bit.

6.3 Trace

With SimpleScalar, we first tested only the return-address protection by running a number of small program traces which were developed to test the function of the Secure Bit, and selected program traces from the SPEC benchmark (e.g. GCC). All successfully ran on top of our SimpleScalar implementation. We found no evidence of new hazards or increased cache latency from our out-of-order simulation. Since testing the function-pointer protection is more complicated, we only validate it against our developed kernel.

7 Cost Analysis

In this section, we analyze several cost aspects of Secure Bit. These aspects include: Backward Compatibility, Deployment, Space (Hardware Requirement), and Performance.

7.1 Backward Compatibility

The Secure Bit protection mechanism is compatible with all legacy binaries. In fact, we have successfully run several serious benchmarks, such as GCC and Apache, in the emulator without any modification. Only the kernel modules required small modifications (as noted above) to enforce such mechanism. We conclude that Secure Bit provides a 100 percent backward compatibility and that the protection mechanism is transparent to the system.

7.2 Deployment

Though the Secure Bit requires new hardware, the amount of investment is relatively small compared to new, security-enhanced hardware such as Intel LaGrande (with Microsoft NGSCB). In fact, Secure Bit is complimentary to LaGrande by enhancing it to provide buffer-overflow protection so Secure Bit could be added onto LaGrande. An intermediate implementation path is also available because Secure Bit could be implemented as middleware. Yet another alternate path is possible by implementing Secure Bit only on the processor with a special segmentation scheme, thus freeing memory from modification (but at a cost to backward compatibility).

7.3 Space

The one-time cost of Secure Bit is comparable to that of a parity bit: having *n* words of memory, an additional *n* bits are needed. From a memory point of view, this Secure Bit is simply another data bit. From a processor point of view, it is only controlled by selected instructions. The Secure Bit need not be kept on disk, but the memory bus will have to handle it. The details of this cost are critical to the success of Secure Bit, and our investigation of an implementation at the processor level with SimpleScalar addresses this point—there are no hidden costs.

7.4 Performance

Accessing Secure Bit adds no additional latency to accessing other data, since Secure Bit access can be done in parallel with accessing its associated data. Few instructions added to the kernel for switching the *sbit_write* mode is also too small to measure. Only trivial additional penalty is additional cycle for rendering the Secure Bit when swapping the page in and out from swapping spaces. However, a hardware/software optimization can play a role

here, but that study is beyond our scope. Thus, ***we can conclude that no performance penalty is introduced.***

8 Conclusions

The necessary and sufficient condition for a buffer-overflow attack is the corruption of an address. Based on that condition, we have proposed, demonstrated, and proven Secure Bit2: a hardware buffer-overflow-attack prevention scheme. Our Secure Bit architecture is transparent to software so it provides a 100% backward compatibility. In addition, it protects not only against first-generation stack smashing, but also against the more recent function-pointer attacks and Global Offset Table attacks.

Since its protection mechanism is based on the principle of protecting the integrity of an address, it should provide protection against future buffer-overflow attacks. Though this paper is limited to an attack that modifies the program control flow, we can apply the concept of the Secure Bit to prevent a type of buffer-overflow attack that modifies a data pointer to leak confidential information (e.g. password, protected data in an object) by preserving the integrity of a data pointer (modifying the syntax of every instruction that accesses memory in indirect mode to validate the Secure Bit).

Furthermore, our booting of Linux on an emulation of Secure Bit demonstrates the transparency and the robustness of our approach. Running real software such as GCC and Apache on that Linux plus staging attacks on the emulation further demonstrates its effectiveness. Finally, our hardware simulation demonstrates Secure Bit's viability at the register-transfer level.

REFERENCES

1 C. Schmidt, and T.Darby, “The What, Why, and How of the 1988 Internet Worm,” *http://www.snowplow.org/tom/worm/worm.html*

2 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

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

4 E. chien and P. Ször (Symantec Security Response), “Blended Attacks Exploits, Vulnerabilities and Buffer-Overflow Techniques in Computer Viruses,” Virus Bulletin Conference, (Sept. 2002)

5 Aleph One, “Smashing stack for fun and benefit,” Phrack Magazine, 49(7), Nov.1996

6 J. Viega, G. McGraw, “Buffer Overflows,” Chapter 7, Building Secure Software, Addison Wesley, pp.135-185 (2002)

7 “OpenSSL Security Advisory [30 July 2002],” *http://www.openssl.org/news/secadv_20020730.txt*

8 S. HsiangRen, “Apache/mod_ssl (slapper) Worm,” GIAC Certified Incident Handler (GCIH), *http://www.giac.org/practical/HsiangRen_Shih_GCIH.doc* (Oct. 2002)

9 D. Geer, “Just How Secure Are Security Products?,” *IEEE Computer Vol.37, No. 6* - June 2004, pp. 14-16

10 “Microsoft Security Bulletin MS04-028: Buffer Overrun in JPEG Processing (GDI+),” *http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx* (Oct, 2004)

11 C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *the proceedings of the 7th USENIX Security Symposium* (Jan. 1998)

12 C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting Systems from Stack Smashing Attacks with StackGuard," *the Linux Expo*, Raleigh, NC (May 1999)

13 C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," *DARPA Information Survivability Conference and Expo (DISCEX)*, Hilton Head Island SC, (Jan. 2000)

14 H. Hinton, C0 Cowan, L. Delcambre, and S. Bowers, "SAM: Security Adaptation Manager," *the Annual Security Applications Conference (ACSAC)*, Phoenix, AZ, (Dec. 1999)

15 T. Krazit, "PCWorld - News - AMD Chips Guard Against Trojan Horses," IDG News Service, *http://www.pcworld.com/news/article/0%2Caid%2C114328%2C00.asp* , (Jan. 14, 2004)

16 Solar Designer, "Linux kernel patch from the Openwall Project (Non-Executable User Stack)," *http://www.openwall.com/*

17 PaX Team , "Documentation for the PaX project," *http://pax.grsecurity.net/docs/index.html*

18 Ingo, "Exec Shield, new Linux security feature," *http://redhat.com/~mingo/exec-shield/*

19 S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits," *Proceedings of the 12th USENIX Security Symposium*, Washington, D.C. (Aug. 2003)

20 J. Etoh., "GCC extension for protecting applications from stack-smashing attacks," *http://www.trl.ibm.com/projects/security/ssp/* (Jun. 2000)

21 L. Torvalds, “Re: [PATCH] [SECURITY] suid procs exec'd with bad 0,1,2 fds,” NEWS Archive, *http://old.lwn.net/1998/0806/a/linus-noexec.html* (Aug. 1998)

22 Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle, “PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities,” *Proceedings of the 12th USENIX Security Symposium*, Washington DC (Aug. 4-8, 2003)

23 Z. Shao, Q. Zhuge, Y. He, E. H.-M. Sha, “Defending Embedded Systems Against Buffer Overflow via Hardware/Software,” *Proceeding of the 20th Annual Computer Security Applications Conference*, Tucson, Arizona (Dec. 6-10, 2004)

24 N. Tuck, B. Calder, and G. Varghese, “Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow,” *Proceedings of the 37th International Symposium on Microarchitecture*, (Dec. 2004)

25 Vendicator, “Stack Shield technical info file v0.7”, *http://www.angelfire.com/sk/stackshield/index.html*

26 J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, “Architecture Support for Defending Against Buffer Overflow Attacks”, *Workshop on Evaluating and Architecting Systems for Dependability* (Oct. 2002)

27 J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee, “A Processor Architecture Defense against Buffer Overflow Attacks”, *Proceedings of the IEEE International Conference on Information Technology: Research and Education (ITRE 2003)*, pp. 243-250, (Aug, 2003)

28 H. Ozdoganoglu, T.N. Vijaykumar, C.E. Brodley, A. Jalote, and B. A. Kuperman, “SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address,” (Nov.22, 2003)

29 A. Baratloo, N. Singh and T. Tsai, “Transparent Run-Time Defense Against Stack Smashing Attacks,” *Proceedings of the USENIX Annual Technical Conference* (2000)

30 M.S. M. Frantzen, “StackGhost: Hardware facilitated stack protection,” *Proceedings of the 10th USENIX Security Symposium* (Aug. 200)

31 T. Chiueh, F. Hsu, “RAD: A Compile-Time Solution to Buffer Overflow Attacks,” *International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, USA (Apr. 2001)

32 M. Prasad, and T. Chiueh, “A Binary Rewriting Defense against Stack based Buffer Overflow Attacks,” *Usenix Annual Technical Conference*, General Track, San Antonio, TX (Jun. 2003)

33 Jones, R. W. M. and Kelly, P.H.J. “Backwards-compatible bounds checking for arrays and pointers in C programs”, *Third International Workshop on Automated Debugging*, Linkoping University Electronic Press (1997)

34 “Rational PurifyPlus,” *http://www-3.ibm.com/software/awdtools/purifyplus/*

35 “BoundsChecker,” *http://www.compuware.com/products/devpartner/bounds.htm*

36 Robert P. Colwell, et al, “Instruction Sets and Beyond: Computers, Complexity and Controversy,” *IEEE Computer*, 18(9) (1985)

37 Organick, Elliott, *A programmer's View of the Intel 432 System*, McGraw-Hill, New York, N.Y. (1983)

38 U.S. Department of Energy Computer incident Advisory Capability, “O-130: Perl and ActivePerl win32_stat Buffer Overflow,” *http://www.ciac.org/ciac/bulletins/o-130.shtml* (Apr. 29, 2004)

39 Sun Alert Notification, “Document ID 57643: Netscape NSS Library Vulnerability Affects Sun Java Enterprise System,” *http://sunsolve.sun.com/search/document.do?assetkey=1-26-57643-1* (Sep. 16, 2004)

40 D. Dean, E. W. Felten, and D. S. Wallach. “Java Security: From HotJava to Netscape and Beyond,” *In Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA (1996)

41 Intel Corporation, “LaGrande Technology Architectural Overview” (Sept, 2003)

42 Microsoft Corporation, “The Next-Generation Secure Computing Base: An Overview,” *http://www.microsoft.com/resources/ngscb/default.mspx*

43 R. MacDonald, S. W. Smith, J. Marchesini, O. Wild, “Bear: An Open-Source Virtual Secure Coprocessor based on TCPA,” T*echnical Report TR2003-471*, Department of Computer Science, Dartmouth College (Aug. 2003)

44 Trusted Computing Platform Alliance, “TCPA IT White paper,” *http://www.trustedcomputing.org*

45 ARM, “TrustZone Technology,” *http://www.arm.com/products/CPUs/arch-trustzone.html*

46 G. S. Kc, A. D. Keromytis, and V. Prevelakis, “Countering Code-Injection Attacks With Instruction-Set Randomization,” *Proceeding of the 10th ACM Conference on Computer and Communications Security*

47 D. Kirovski, M. Drinic, and M. Potkonjak., “Enabling Trusted Software Integrity,” *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2002)

48 G. C. Necula, S. McPeak, and W. Weimer, “CCured: Type-Safe Retrofitting to Legacy Code,” *The Proceedings of the Principles of Programming Languages* (2002)

49 John Viega, J.T. Bloch, Yoshi Kohno, Gary McGraw, “ITS4: A Static Vulnerability Scanner for C and C++ Code,” *Proceeding of the 16th Annual Computer Security Applications Conference*, New Orleans, Louisiana (Dec. 2000)

50 Flawfinder, *http://www.dwheeler.com/flawfinder/*

51 RATS, http://www.securesw.com/rats/

52 D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," *IEEE Software* (Jan.-Feb. 2002)

53 E. Haugh, M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," *Proceedings of the 2003 Symposium on Networked and Distributed System Security (SNDSS 2003)* (Feb. 2003)

54 D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," *Proceeding of the 10th USENIX Security Symposium* (Aug. 2001)

55 C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman, "FormatGuard: Automatic Protection From printf Format String Vulnerabilites," *USENIX Security Symposium*, Washington DC (Aug. 2001)

56 U. Shankar, K.Talway, J.S. Foster, and D. Wagner, "Detecting Format String Vulnerabilities with Type Qualifiers," *Proceedings of the 10th USENIX Security Symposium*, pp. 201-216 (Aug. 2001)

57 J. Wilander and M. Kamkar, "A Comparison of Publicly Available tools for Static intrusion Prevention," *7th Nordic Workshop on Secure IT systems*, Karlstad, Sweden (2002)

58 J. Wilander and M. Kamkar, "A Comparison of Publicly Available tools for Dynamic intrusion Prevention," *10th Network and Distributed system Security Symposium (NDSS)* (2003)

59 Bulba and Kil3e, "Bypassing stackguard and stackshield," *Phrack Magazine*, 10(56) May 2002.

60 T. Newsham, "BugTraq Archive: Re: StackGuard: Automatic Protection From Stack-smashing Attacks," http://www.securityfocus.com (Dec. 19, 1997)

61 D. Litchfield, “Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server,” NGSSoftware Ltd, *http://www.nextgenss.com* (Sep. 2003)

62 F. Chang, A. Itzkovitz, V. Karamcheti, “User-level Resource-constrained Sandboxing,” *USENIX Windows System Symposium* (Aug. 2000)

63 D. S. Peterson, M. Bishop, R. pandey, “Flexible Containment Mechanism for Executing Untrusted Code,” *Proceedings of the 11th USENIX UNIX Security Symposium*, pp. 207-225 (Aug. 2002)

64 M. Bishop, Computer Security, Addison-Wesley, (Dec. 2002)

65 “Bochs IA-32 Emulator Project,” *http://bochs.sourceforge.net/*

66 “SimpleScalar,” *http://www.simplescalar.com/*

67 Webopedia Computer Dictionary, “What is buffer overflow?”, *http://webopedia.com/TERM/b/buffer_overflow.html*

68 F, Swiderski and W. Snyder, Threat Modeling, Microsoft Press (2004)

69 K. Piromsopa, M. Fletcher, and R. Enbody, “Secure Bit : Hardware, Buffer-Overflow Prevention,” Technical Reports #MSU-CSE-04-48, Department of Computer Science and Engineering, Michigan State University (2004)

70 K. Piromsopa and R. Enbody, “Buffer Overflow: Fundamental,” Technical Reports #MSU-SE-04-47, Department of Computer Science and Engineering, Michigan State University (2004)

Appendix

This section lists the modification

For “uaccess.h”, two macros are added: SET_SBITMODE, and CLR_SBITMODE.

```
// For Secure Bit 2
#define SET_SBITMODE() \
    asm volatile(                          \
        "    pushl    %eax\n"    \
        "    lahf\n"    \
        "    orb    $0x20, %ah\n"    \
        "    sahf\n"    \
        "    popl    %eax"    )

#define CLR_SBITMODE() \
    asm volatile(                          \
        "    pushl    %eax\n"    \
        "    lahf\n"    \
        "    andb    $0xdf, %ah\n"    \
        "    sahf\n"    \
        "    popl    %eax"    )
```

For “usercopy.c”, four functions are modified:

- __generic_copy_to_user
- __generic_copy_from_user
- __strncpy_from_user
- strncpy_from_user

```
unsigned long
__generic_copy_to_user(void *to, const void *from, unsigned long n)
{
        SET_SBITMODE();
        if (access_ok(VERIFY_WRITE, to, n))
                __copy_user(to,from,n);
        CLR_SBITMODE();
        return n;
}


unsigned long
__generic_copy_from_user(void *to, const void *from, unsigned long n)
{
        SET_SBITMODE();
        if (access_ok(VERIFY_READ, from, n))
                __copy_user_zeroing(to,from,n);
        CLR_SBITMODE();
        return n;
}
```

```
long
__strncpy_from_user(char *dst, const char *src, long count)
{
        long res;
        SET_SBITMODE();
        __do_strncpy_from_user(dst, src, count, res);
        CLR_SBITMODE();
        return res;
}


long
strncpy_from_user(char *dst, const char *src, long count)
{
        long res = -EFAULT;
        SET_SBITMODE();
        if (access_ok(VERIFY_READ, src, 1))
                __do_strncpy_from_user(dst, src, count, res);
        CLR_SBITMODE();
        return res;
}
```