Secure Bit enhanced Canary: hardware enhanced buffer-overflow protection.

Krerk Piromsopa
Department of Computer Engineering
Chulalongkorn University
Bangkok 10330, Thailand
krerk@cp.eng.chula.ac.th

Sirisara Chiamwongpaet
Department of Computer Engineering
Chulalongkorn University
Bangkok 10330, Thailand
u47sch@cp.eng.chula.ac.th

Abstract

Piromsopa and Enbody[10] proposed Secure Bit[1] , an architectural approach to protect against buffer-overflow attacks on control data (return-addresses and function-pointers). This paper explores the possibility of extending Secure Bit to protect non-control data (variables, pointers and arrays). A hardware bit, provided by Secure Bit, helps preserving the integrity of an associated address. We propose putting a Canary Word adjacent to variables and using Secure Bit to protect the integrity of this word. In this extension, we introduce a new hardware instruction that is used to validate the Secure Bit of an adjacent word. An important differentiating aspect of Secure Bit is that once a Canary Word (data) has been marked as insecure there is no instruction to remark it as secure. Thus, it is theoretically not possible to bypass the mechanism by overflowing the Canary Word with a valid value. Our preliminary study has shown a promising solution to buffer overflow on arbitrary data. Robustness and performance are demonstrated by emulating the hardware, booting Linux on the emulator, running application software on that Linux, and performing known attacks.

1. Introduction

Secure Bit[10] is originally designed to protect buffer-overflow attacks on control data (return address and function pointers). We extend Secure Bit to protect buffer-overflow attacks on data (variables, pointers and arrays). The goal is to apply the strength foundation of Secure Bit to prevent buffer-overflow attacks on variables.

Date back to the infamous MORRIS worm of 1988[12], buffer-overflow attacks remain the most common. Skilled programmers can write code without buffer-overflow vulnerabilities. However, no program is guaranteed free from bugs. As buffer overflows are eliminated in operating systems, they are being found and exploited in applications. When applications are run with root or administrator privileges the impact is equally devastating.

We assume that Secure Bit works—as demonstrated and proved by Piromsopa and Enbody[10, 11, 9]. Since Secure Bit is relatively new, we begin with a brief overview. The underlying concept is to keep track of data controlled by users and revert it from being used as control data. The mechanism can be sum up as:
"data passing from another domain must not be used as a return address or a function pointers."

To enforce this condition and maintain transparency to legacy user code, Secure Bit associates a hardware bit to each memory word (or byte). This "Secure Bit" serves as integrity metadata: any word with Secure Bit set is an untrustworthy word. Control instructions (call, jump, and return) are modified to prevent the processor from using untrustworthy addresses as target addresses. Since this hardware bit is not visible to user, Secure Bit is transparent to application (binary backward compatibility).

Though Secure Bit sounds promising, it shares the same drawback of almost all buffer-overflow prevention schemes: overflow to non-control data is not protected. This limitation is actually a strength in that it provides full protection against a whole class of attacks rather than limited protection against a broader class of attacks.

Assessment of Secure Bit unveils that protecting non-control data cannot intuitively be enforced and maintain transparency to legacy user code. While control data tends to be created locally without deriving any data from another domain (i.e. return address is created locally by call instruction), variables are usually used to hold input or data derived from input. Thus, the Secure Bit protocol cannot be applied directly to transparently protect non-control data.

[1]Patent Pending

978-0-7695-3354-4/08 $25.00 © 2008 IEEE
DOI 10.1109/NPC.2008.49

Note that there exists a dynamic tainted solution [3] that try to apply similar architecture to protect pointers by allowing program to untainted verified data. However, such high ambition allows a new vector of attack on untainted data and results in an incomplete solution.

One strength property is that once the associated data is tainted, it cannot be untainted. The characteristic makes Secure Bit difficult to bypass (if not impossible) [9].

To apply Secure Bit to non-control data, we explore alternate metadata that can be used to detect a buffer overflow. There exists a Canary solution (such as StackGuard [4] and Windows 2003[6] that can be applied to protect non-control data. By placing a Canary Word adjacent to each variable, validating the value of this Canary Word before using the variable would indirectly detect buffer-overflow attacks. However, Canary solution in itself is a weak solution in that it is possible to overflow using a valid Canary [1]. Though some researchers[5] suggest using a per-process Canary which is generated dynamically when a program starts, these methods only reduce the probability of successful attacks.

To strengthen the Canary solution, we enforce Secure Bit protocol on a Canary Word. Since Secure Bit provides mechanism for differentiating value created locally and value derived from data passing from another domain, overflowing with a valid Canary value is immediately impractical.

The goal of this paper is to enhance Canary Solution by applying Secure Bit. Before discussing Secure Bit and Canary Solution, we begin with background on buffer-overflow attacks. Later, we evaluate performance and effectiveness of our solution. We conclude that Secure Bit has a potential as a hardware enhanced buffer-overflow protection for non-control data.

2. Concept

This section lays the ground work for understanding buffer overflow, buffer-overflow attacks and buffer-overflow protections.

2.1. Buffer Overflow

There is no formal definition of buffer overflow that we can use. So, we took definition from several places (e.g. [14] and [9]) and defined it for our purpose.

Definition 1: *A* ***buffer overflow*** *is the condition wherein the data transferred to a buffer exceeds the storage capacity of the buffer and some of the data "overflows" into another buffer, one that the data was not intended to go into.*

From this definition, the data can be replaced not only by overflowing buffer, but also by writing beyond data boundary, which will affect the computer system. Therefore, we define a buffer-overflow attack as follow:

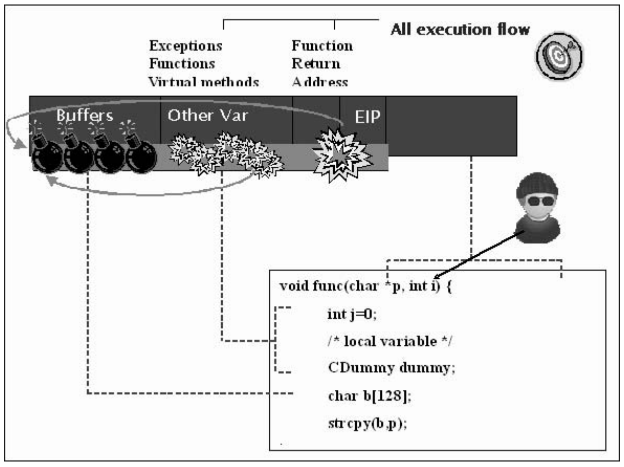
Definition 2: *A* ***buffer-overflow attack*** *is an attack caused by overflowing a buffer or writing beyond data boundary with data from another domain which results in malicious or unexpected behavior of a program.*

This is considered violation of data's integrity. Moreover, the replaced data can be any data in the memory (either control data or non-control data). To ease understanding, here is an example of buffer-overflow attacks.

Example 1: Stack-smashing Attack

Figure 1 shows stack-smashing attack. Stack smashing happens when an attacker (shown in the black hat) passes a buffer containing malicious code (e.g. shell) and multiple copies of address of the target buffer as an argument to a vulnerable program (through parameter *p*). When running to a buffer manipulation function (e.g. `strcpy` in this example), the overflow data will replace the function's return address with the address of the target buffer (containing malicious code). The eventual result is that the return instruction will use the address of the target buffer as a return address and redirect the program flow to execute injected malicious code.

Stack-smashing attack is the most common and well-studied buffer-overflow attacks. It primarily targets control data (return address) by replacing it with addresses of the target buffer.



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

Figure 1. Stack-smashing Attack

In practical, there are several variations of buffer-overflow attacks. Classifying by a location where the overflow occurs, buffer-overflow attacks can be classified into three categories: stack overflow, heap overflow and array indexing error.

Stack overflow is an attack caused by overflowing a buffer in the stack area of a process. Though it may affect control data (i.e. stack-smashing attack) or non-control data, control data (return address) is generally a target. Stack smashing is among the most common type.

Heap overflow is a type of buffer overflow that occurs in the heap area of a process. Since heap keeps dynamic allocation data (e.g. memory area reserved by `malloc` function) in run-time environment, an attacker can inject malicious code in heap and redirect the program flow to execute the malicious code in this area as well.

Array indexing error differs from stack overflow and heap overflow in that it is caused by indexing beyond the boundary of array. It occurs by modifying the index value to be out of boundary of the associated array. Given the possibility of indexing value, it is theoretically possible to write to arbitrary memory location. Pointer arithmetic also falls in this category. In another word, attackers can create arbitrary pointers.

2.2. Buffer-overflow Protection

Up to this point, we have establish a rudimentary understanding of buffer-overflow attacks. In this section, we focus on the protection schemes. There exist several approaches to buffer-overflow attacks. Piromsopa [8] has established a taxonomy of buffer-overflow protection schemes. With this taxonomy, protection schemes can be partitioned into 3 broad categories: static analysis, dynamic solutions, and isolation (Shown in Figure 2). Static analysis is a class of solutions that parses source code and suggests programmers to prevent the problem before deploying the program (e.g. semantic analysis). Dynamic solutions refer to schemes that create metadata for validating integrity of data during the execution of programs. Isolation is a class of solution that minimize the damage causing by attacks (e.g. sandboxing).

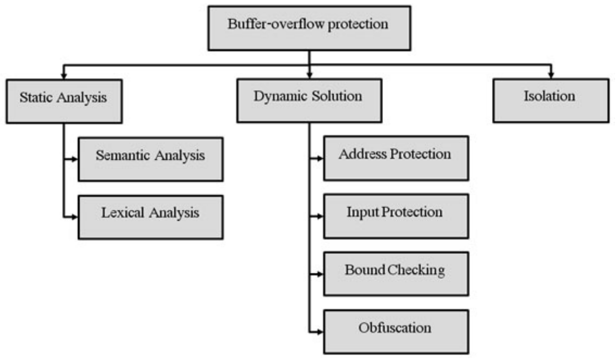


Figure 2. Taxonomy of solutions against buffer-overflow attacks

We are particularly interested in dynamic solutions. Depending on the underlying assumptions, solutions in this class vary in the management of metadata and handling routines. Nonetheless, they share the same idea of using metadata obtaining from the run-time environment for validating the integrity of data. Based on the study of Piromsopa [8], dynamic solutions can also be further classified by the management of metadata into bounds checking, obfuscation, address protection, and input protection. In bounds checking scheme, every access to memory must be specified as base and limit (aka. segment descriptors). Though it sounds promising, this mechanism is not common in every architecture and usually disabled in supported architecture for performance reason. Segmentation [7] is an example of solutions in this scheme. Obfuscation referred to solutions that reduces the probability of successful attacks using randomness (i.e. randomly insert space between variables). PAX [13] serves as a good example.

To our purpose, we will further elaborate address protection and input protection. In particular, we will use Canary Word and Secure Bit as an example of each protection scheme respectively.

2.2.1 Address Protection

Given that addresses are the target, solutions in this class creates a metadata for validating the integrity of addresses before using it. Assuming that corrupting an address will also corrupt the adjacent data, Canary Word solutions (e.g. StackGuard [4]) place a metadata (known Canary value) adjacent to an address. By validating the Canary value, it is possible to detect a buffer overflow.

The general mechanism can be described as:
1. Place a Canary Word next to the address (e.g. return address) when the address is created (e.g. by a call instruction).
2. Verify the Canary Word before using the address (e.g. before a return instruction uses a return address).

To understand the effectiveness and the weakness of this Canary solution, we show example 2.

Example 2: Canary Word

Given the following code, a Canary will be a word inserted (possibly by compiler) between *buffer* and *b* in the stack area.

```
int main(char argc, char *argv)
{
    char b = '0';
    char buffer[3];
    strcpy(buffer, argv[1]);
    printf("%d", b);
    ...
    return 0;
}
```

Passing a string `"AAAAA"` to this program would overflow the variable *b* with a character `"A"`. Assuming that the Canary Word is originally constructed as `"0"`, the value `"A"` in the Canary Word would indicate that there is an overflow. However, bypassing this protection scheme can intuitively be done by replacing a Canary Word with a valid Canary value. For example, a string `"AAAA0A"` would allow attackers to change the value of variable *b* without setting the alarm. Table 1 shows such stack layout in different conditions.

Table 1. Stack layout of the example 2

Before buffer overflow	Type	Buffer			Canary Word	b
	Value	-	-	-	0	5
After buffer overflow	Type	Buffer			Canary Word	b
	Value	A	A	A	A	A
Bypassing Canary Word	Type	Buffer			Canary Word	b
	Value	A	A	A	0	A

2.2.2 Input Protection

This method assumes that input should be treated differently from local data (i.e. input should not be used as control data). Secure Bit is a good example of solutions in this class. Secure Bit uses a hardware bit (Secure Bit) associated with each memory word for tracking data passing across domains (process and kernel). If a bit is set, the associated data is either an input or derived from input and should not be used as control data (return address, and function pointers). We briefly discuss Secure Bit in example 3 and point out the failure in protecting non-control data in example 4.

Example 3: Secure Bit and control data

Given the following code, attackers may replace a return address by passing a string longer than three characters as a command-line argument into the program. With Secure Bit in place, data passing across domains would have the Secure Bit set. Thus, the system would be able to validate that the return address is not created locally. In Secure Bit enhanced architecture, this validation is embedded in the semantic of associated instructions (e.g. call, return, and jump). Table 2 show the stack layout before and after the attack.

```
int main(char argc, char *argv)
{
    char buffer[3];
    strcpy(buffer, argv[1]);
    ...
}
```

Example 4: Secure Bit and non-control data

We will demonstrate that Secure Bit cannot be applied directly to non-control data. If `b` is also derived from input, the Secure Bit of variable *b* would fail to identify buffer overflow. The memory snapshot is shown in Table 3.

```
int main(char argc, char *argv)
{
    char b;
    char buffer[3];
    b = argc;
    strcpy(buffer, argv[1]);
    ...
}
```

We have learned that buffer-overflow attacks can happen to both control and non-control data. Nevertheless, most buffer-overflow protections focus only on control data. Though some solutions render buffer-overflow attacks on control data impossible (or relatively difficult), buffer-overflow attacks on non-control data (e.g. variables) has emerged as a new generation of attacks.

3. Design and Implementation

We proposed using a combination of Canary Word and Secure Bit, namely “Secure Canary Word”, for preventing buffer-overflow attacks on non-control data.

3.1. The principle of Secure Canary Word

The principle of Secure Canary Word (Secure Bit enhancing Canary Word) is to conceal the weakness of the Canary Word using the integrity provided by Secure Bit. While a Canary Word can be inserted to detect buffer overflow of data, there is no mechanism to protect the Canary Word itself from being overflowed. Thus, Secure Bit can

Table 2. Stack layout of the example 3

Before buffer overflow	Type	Buffer			Return address
	Value	-	-	-	5
	Secure Bit	0	0	0	0
After buffer overflow	Type	Buffer			Return address
	Value	A	A	A	A
	Secure Bit	1	1	1	1

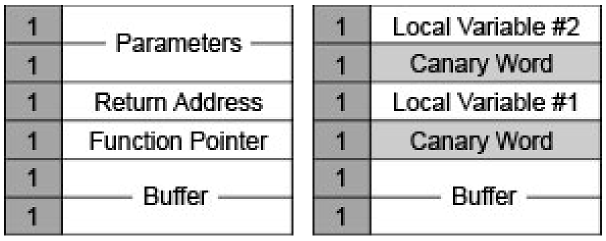
Table 3. Stack layout of the example 4

Before buffer overflow	Type	Buffer			b
	Value	-	-	-	5
	Secure Bit	0	0	0	1
After buffer overflow	Type	Buffer			b
	Value	A	A	A	A
	Secure Bit	1	1	1	1

be used for detecting an overflow of the Canary Word. (i.e. Overflowing with a valid Canary value can be detected by Secure Bit). In addition to buffer-overflow attacks on control data, we expect that the proposed method allows us to extend Secure Bit to protect non-control data. Table 4 shows the stack layout of code in example 2 when presented in the Secure Canary Word architecture. Figure 3 shows a comparison between Secure Bit and Secure Canary Word.

Table 4. Stack layout of the example 2

Before buffer overflow	Type	Buffer			Canary Word	b
	Value	-	-	-	0	5
	Secure Bit	0	0	0	0	0
After buffer overflow	Type	Buffer			Canary Word	b
	Value	A	A	A	0	A
	Secure Bit	1	1	1	1	1



(a) Secure Bit (b) Secure Canary Word

Figure 3. (a)Secure Bit vs (b)Secure Canary Word

3.2. Implementation

To validate our design, we extends BOCHS[2], i368 emulator, to support our scheme. (The Secure Bit implementation is taken from[10].) There are two modifications necessary for Secure Canary Word: adding new instruction for validating the Secure Bit of the Canary Word (Secure Canary Word), and modifying software to use this instruction for validating the Secure Bit associated with the Canary Word.

1. Adding new CPU instruction

We define a new instruction using unused opcode in the i386 architecture (`OxOf Ox1a`). This instruction is responsible for validating the Secure Bit of register EAX in the processor. If the Secure Bit is cleared, register EAX will be cleared. In contrast, register EAX will be set to 255 if the Secure Bit is set.

2. Software Modifications

The ultimate target is to modify the compiler to used the instruction provided by the underlying architecture to protect data against buffer-overflow attacks. To validate our design, we create three new functions for validating the Secure Bit as followed:

1) `chkSBit` function

This function is serving as a programming interface between C programming language and underlying architecture. We implement this function using in-line assembly. This function basically allows us to validate Secure Bit of any variable. `sbit`. If the value of Secure Bit is not ‘0’, the function will display a warning message on the screen. The code is shown below. Notice that we use `inline` specifier for optimization.

```
inline void chkSBit(char canWord)
{
    int sbit = 0;
    asm volatile
    ("
        movl %1,%%eax;
        .short 0x1A0F;
        movl %%eax,%0;
    "
        :"=m"(sbit)
        :"m"(canWord)
        :"%eax"
    );

    if(sbit != 0)
        printf("Warning: The following
        variable is modified abnormally!");
}
```

2) `read_int` function

This function is created as a wrapper for accessing an integer variable. The idea is to validate the Canary Word adjacent to the variable (using `chkSBit` function) and immediately return the value of the associated variable. With this function in hand, a statement `int_a=int_b;` will be changed to `int_a=read_int(int_b);`. This way, an overflow to variable `int_b` will be detected in the underlying architecture. The detail is described as followed:

```
inline int read_int(int *data)
{
  char *p = data;
  char canWord;
  p--; /* aligning for Canary Word */
  canWord = *p;
  chkSBit(canWord);
  return *data;
}
```

3) `read function`

This function is similar to the `read_int` function. It is, however, specifically customized for validating Secure Canary Word that is associated with a pointer.

```
inline void *read(void *data)
{
    char *p = data;
    char canWord;
    p -= 4; /* aligning for Canary Word */
    canWord = *p;
    chkSBit(canWord);
    return data;
}
```

These three functions allow a program to easily be modified to use Secure Canary Word. The idea is to allocate a Canary Word adjacent to every variable and use the function `chkSBit` to validate the integrity of this Canary Word. For integer and pointer, `read_int` and `read` function will be particularly useful to insert directly into the program. For example,
`a = read_int(b);`
and
`read(buffer);`

4. Experiments

Our experiments is conducted in the modified BOCHS emulators running Linux (Red Hat 6.2). We evaluate our architecture in 2 aspects: protection and efficiency.

4.1. Protection

Initially, we test three types of buffer-overflow attacks: Stack overflow on control data, Stack overflow on non-control data and Array indexing error. Given that Secure Bit is strong enough to protect all (most) control data, we believe that these classes of attacks is enough to cover a wide variety of buffer-overflow attacks on non-control data (variables and pointers).

As expected, Secure Bit transparently flagged buffer-overflow attacks on control data. In addition, our Secure Canary Word allows us to catch buffer-overflow attacks on variables that is missed by the original Canary Word and Secure Bit scheme. Like others, most solutions (including our solution) are still suffer from the Array Indexing Errors and pointer arithmetic where attackers indexing directly to modify target variables.

4.2. The efficiency

We measure the efficiency of Secure Canary Word using 3 simple programs: bubble sort, quick sort and binary search using AVL tree. Among the three program, the AVL tree program is a pointer intensive program. The original programs (without Secure Canary Word) are baselines in our comparison. We compare the execution time and the size of the program binary between the original version and our Secure-Canary-Word enhanced version. Programs are written in C with optimization flag (`-O3`) turn on. Table 5 and Table 6 shows the execution time and the program size respectively. Since our study is still in early stage. It is worth clarifying that these experiment does not taken into account any redundant operations (global optimization).

Table 5. Execution Time

Example Programs	Execution Time(Seconds)		
	Before	After	Ratio
Bubble sort	2.88	336.34	+116.78
Quick sort	1.14	6.55	+5.75
Binary search using AVL tree	1.04	20.83	+20.03

4.3. Evaluation

The results show that the Secure-Canary-Word enhanced programs are much more slower than the ordinary programs. Since there is no global or hardware optimization involved in our experiments (i.e. every access to the memory verifies the adjacent Canary Word), the slow down is not surprised. In addition, the size of programs are dramatically increased, especially in the pointer intensive program (AVL tree). This is directly a result from the use of `inline` specifier. Thus, every call our functions (`chkSBit`, `read_int`, and `read`) is expanded into a chunk of code.

To address this issue, we are now investigating hardware optimization to reduce both execution time and program size. A possible solution is to introduce an instruction for reading data and verifying the Secure Bit of associated Canary Word in parallel. (This instruction would powerful enough to replace `chkSbit`, `read_int` , and `read` functions with 1 assembly instruction executing in 1 CPU clock cycle.) With this design, the overhead will be buried deep in the hardware. Once our hardware enhanced design is complete, we believe that increasing in code size will be limited to the additional space for Canary Word. Comparing to bounds checking solutions (both hardware and software), we believe that metadata required to archive the same goal is minimal. While a descriptor requires base address and

Table 6. Size of program

Example Programs	Size(Bytes)		
	Before	After	Difference(%)
Bubble sort	17,193	22,531	+31.05
Quick sort	18,268	29,838	+63.33
Binary search using AVL tree	27,072	44,559	+64.59

offset (at least 2 words), Secure Canary Word requires only one byte (or word) per variable.

5. Conclusions

The principle of Secure Bit in preserving integrity of data across domains provides a solid protection against buffer-overflow attacks on control data. By preserving integrity of a word adjacent to a variable, Secure Bit can be extended to provide protection for non-control data (pointers and variables).

This work is still in the initial stage. There exist several opportunities for hardware and software optimization. In conclusion, our study has shown the potential of Secure Bit architecture as a platform for providing buffer-overflow protections on variables and data.

References

[1] Bulba and Kil3e. Bypassing stackguard and stackshield. *Phrack Magazine*, 10(56), 2000.

[2] T. R. BUTLER. Welcome to the bochs ia-32 emulator project. http://bochs.sourceforge.net/, 2006.

[3] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer. Defeating memory corruption attacks via pointer taintedness detection. *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 378–387, June-1 July 2005.

[4] C. Cowan, S. Beattie, R. Day, C. Pu, P. Wagle, and E. Walthinsen. Protecting systems from stack smashing attacks with stackguard. In *Linux Expo*, Raleigh, NC, 1999.

[5] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: protecting pointers from buffer overflow vulnerabilities. In s, editor, *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 7–7, Berkeley, CA, USA, 2003. USENIX Association.

[6] D. LITCHFIELD. Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server. In *NGSSoftware*, 2003.

[7] E. Organick. *A Programmers View of the Intel 432 System*. McGraw-Hill, 1983.

[8] K. Piromsopa. *SECURE BIT: BUFFER-OVERFLOW PROTECTION*. PhD thesis, Michigan State University, 2006.

[9] K. Piromsopa and R. Enbody. Buffer-overflow protection: The theory. *Electro/information Technology, 2006 IEEE International Conference on*, pages 454–458, May 2006.

[10] K. Piromsopa and R. J. Enbody. Secure bit: Transparent, hardware buffer-overflow protection. *Dependable and Secure Computing, IEEE Transactions on*, 3(4):365–376, Oct.-Dec. 2006.

[11] K. Piromsopa and R. J. Enbody. Architecting security: a secure implementation of hardware buffer-overflow protection. In *ACST'07: Proceedings of the third conference on IASTED International Conference*, pages 17–22, Anaheim, CA, USA, 2007. ACTA Press.

[12] C. Schmidt and T. Darby. The what, why, and how of the 1988 internet worm. http://www.snowplow.org/tom/worm/worm.html.

[13] P. TEAM. Documentation for the pax project. http://pax.grsecurity.net/, 2003.

[14] Webopedia. What is buffer overflow? http://www.webopedia.com/TERM/B/buffer_overflow.html.