

CANARY BIT: EXTENDING SECURE BIT FOR DATA POINTER
PROTECTION FROM BUFFER OVERFLOW ATTACKS

By

Michael S. Kirkpatrick

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science and Engineering

2007

ABSTRACT

CANARY BIT: EXTENDING SECURE BIT FOR DATA POINTER PROTECTION FROM BUFFER OVERFLOW ATTACKS

By

Michael S. Kirkpatrick

The buffer overflow attack is one of the oldest and most pervasive vulnerabilities in computer security. Though there have been many advancements in the fields of software engineering and hardware development to defend against known threats, the techniques used in buffer overflow attacks continue to evolve. Consequently, no single mechanism has been able to offer a complete defense against these attacks. *Secure Bit 2* offers a minimalist, architectural approach that successfully protects against buffer overflow attacks on control data. In this work, we introduce *Canary Bit* as an extension of *Secure Bit 2* to provide a flexible technique that protects against buffer overflow attacks on pointers that are not used as control data. In this extension, we create a new hardware instruction that is used to enforce the integrity of the value of a pointer before it can be dereferenced.

ACKNOWLEDGEMENTS

I would like to thank Dr. Richard J. Enbody for serving as my advisor and for his guidance in this endeavor. I also thank Dr. Krerik Piromsopa for sharing his technical expertise and for answering when I would bombard him with questions. Additional thanks go to my committee members, Dr. Anthony S. Wojcik and Dr. Wayne Dyksen, for reviewing this work.

Finally, special thanks go to my wife, Brianne, for her enduring love and boundless patience. I could not have completed this work without her unwavering support. She is my inspiration and the source of all my happiness.

PREFACE

Throughout this work, we will have code interspersed with the text. Our conventions are as follows. If we are using a function name or a variable within the context of a normal paragraph, we will *italicize* the relevant word to distinguish it from the rest of the sentence. On a few occasions, we will include an entire line of code in the text. These lines will be written in a `monospace font`. Sections of code that are longer than a single line will be displayed in a figure and surrounded by a box.

In the Implementation section, we frequently use the term “micro operations” to describe behavior defined within QEMU. If we were defining the term, it would be hyphenated or combined into a single word (*i.e.*, micro-operations or microoperations). However, we have chosen to write the word as it was used in the original QEMU document [1].

TABLE OF CONTENTS

LIST OF TABLES	vii
LIST OF FIGURES	x
1 INTRODUCTION	1
2 BACKGROUND	3
2.1 Related Work	3
2.2 System Overview	4
2.3 Alternate Implementations	6
3 FUNDAMENTALS	9
3.1 Definitions	9
3.2 Limitation of Scope	11
3.3 Formalization of Concept	11
4 IMPLEMENTATION	14
4.1 ARM Architecture Changes	14
4.2 GNU Assembler	16
4.3 Perl Utility Script	18
4.4 Linux	19
4.5 QEMU ARM Emulator	20
4.5.1 QEMU ARM-specific Modifications	21
4.5.2 QEMU Generic Code Modifications	34
5 EVALUATION	41
5.1 Sample Attack Wrapper Code	41
5.2 Stack Smashing Attacks	42
5.3 Unstructured Heap Attacks	44
5.4 Heap Corruption Using <i>free</i>	44
5.5 Using Valid Input Data	48
6 ISSUES	53
6.1 Virtual Memory	53
6.2 Memory Interface	54
6.3 Restriction on the Programming Model	55
6.4 Possible Attacks	55
6.5 Cost Analysis	56
6.5.1 Backward Compatibility	56
6.5.2 Space	56
6.5.3 Performance	57
7 FUTURE WORK	58
8 CONCLUSION	59

LIST OF TABLES

1	Summary of QEMU micro operations and the result for the register's Canary Bit	29
2	Micro operations generated by macros in target-arm/op_mem.h . . .	32
3	Micro operations generated by macros in target-arm/op_mem_raw.h .	34
4	Function prototypes modified in cpu-all.h	36
5	Functions that reference <i>cpu_physical_memory_rw</i>	37
6	Modified debug functions in disas.c, monitor.c, and gdbstub.c	40

LIST OF FIGURES

1	Passing input data to a variable	5
2	Example of allocating heap memory for a struct	7
3	A format string vulnerability	11
4	Assigned CPSR bits	14
5	Format of a load/store word/unsigned byte instruction	15
6	Encoding of operation <code>ldvd r0, [r1, #8]</code>	16
7	Lines added to <code>gas/config/tc-arm.c</code>	16
8	Lines added to <code>opcodes/arm-dis.c</code>	17
9	The <code>ReplaceDir</code> function in the <code>cangecc</code> utility script	18
10	The modified version of <code>arch/arm/lib/sbit_macro.S</code>	20
11	Sample of the modifications made in <code>arch/arm/lib/copy_template.s.S</code>	20
12	Modifications to the <code>CPUARMState</code> structure	22
13	Modifications to <code>cpsr_read</code> and <code>cpsr_write</code> in <code>target-arm/cpu.h</code>	22
14	Clearing the PC Canary Bit on a CPU reset	23
15	Canary Bit additions to switching the CPU mode	23
16	Modifications to the QEMU interrupt handler	24
17	Changes to <code>get_phys_addr</code> in <code>target-arm/helper.c</code>	24
18	Modification of <code>msr_mask</code> in <code>target-arm/translate.c</code>	25
19	Added functionality for translating <code>ldvd</code> in <code>target-arm/translate.c</code>	27
20	Changes made to <code>target-arm/op.c</code> for handling loads to and from QEMU registers	28
21	Examples of the changes made to <code>target-arm/op_template.h</code>	28
22	Moving immediate values to, or swapping values of QEMU registers	29
23	QEMU micro operation to access user registers	30

24	Canary Bit functionality added to target-arm/op_mem.h	31
25	Modifications made to target-arm/op_mem_raw.h	33
26	Existing functionality of <i>op_addl_T1_T2</i> is preserved	34
27	Allocation of space for the Canary Bit memory in vl.c	35
28	The <i>ld_cbit</i> function defined in cpu-all.h	35
29	The <i>st_cbit</i> function defined in cpu-all.h	36
30	The <i>ld_cbit_base</i> function defined in cpu-all.h	36
31	The <i>st_cbit_base</i> function defined in cpu-all.h	36
32	The <i>cpu_physical_memory_rw</i> function in exec.c	38
33	The <i>ldl_phys</i> function defined in exec.c	39
34	The <i>stl_phys</i> function defined in exec.c	39
35	Sample logical flow for a Canary Bit load in softmmu_header.h and softmmu_template.h	39
36	Sample logical flow for a Canary Bit store in softmmu_header.h and softmmu_template.h	40
37	A sample buffer overflow attack program	42
38	The main method for a vulnerable application	42
39	A function vulnerable to a stack-based buffer overflow	43
40	Overflowing a stack variable without Canary Bit	43
41	Canary Bit successfully stopping a stack-smashing attack	43
42	An application vulnerable to a heap-based buffer overflow	44
43	Overflowing a heap variable without Canary Bit	45
44	Canary Bit successfully stopping a heap-based overflow	45
45	The structure of a memory block for allocating heap chunks	46
46	A visual representation of the memory allocation structure	46

47	Replacing pointer values when a chunk is freed	46
48	Our simulated <i>free</i> function	47
49	The function referencing the vulnerable <i>free</i>	48
50	The wrapper functions for the program vulnerable to the <i>free</i> attack .	49
51	Attacking <i>free</i> without Canary Bit	49
52	Canary Bit successfully stopping an attack on <i>free</i>	49
53	Canary Bit permits valid access to legitimate input data	50
54	Demonstrating valid input data without Canary Bit	50
55	Canary Bit does not create a false positive for valid input data	50
56	Different GCC compilations of similar functionality	51
57	Programming work-around for a false positive	55

1 INTRODUCTION

As the role of technology continues to grow, so does the necessity of protecting the stability and integrity of computer systems. One of the oldest and most dominant threats to this security is the buffer overflow vulnerability. These attacks exploit programming flaws that allow a malicious entity to provide manufactured input data that causes the program to behave incorrectly. In its typical form, the buffer overflow attack provides too much data in a place where the program fails to adequately ensure proper bounds. The result is that the program will continue to write this data to memory, overwriting other variables in the process [2]. This can allow an attacker to perform various nefarious acts, including gaining illicit root access, overwriting key kernel variables, destroying files, etc.

Buffer overflow attacks have a long history, dating back to the Morris Worm in 1988 [3]. Other famous examples of viruses and worms that exploit buffer overflows include CodeRed [4], Nimda [4], and Apache Slapper [5]. One difficulty with these attacks is that vulnerabilities can be found in any type of application, including JPEG image processing (GDI+) [6], the Java disassembler [7], Perl file handling [8], common C library functions [9], Microsoft Office 07 [10], the BrightStor ARCserve Backup for Vista [11] and some image readers for e-passports [12]. Even buffer overflow defense mechanisms have been shown to have flaws [13, 14, 15].

Due to the frequent occurrence of buffer overflows and the potential danger that a successful attack poses, there has been a lot of research done on the subject. Pincus and Baker [16] offer a useful taxonomy of the various types of attacks. Though most of the literature on the subject focuses on control data, such as return addresses and function pointers, attacks against non-control data have also been shown to be a viable threat [17]. In this work, we will focus on this latter category.

While most of the available prevention mechanisms offer incomplete solutions, Secure Bit 2 provides a minimalist, architectural approach that offers complete pro-

tection of control data immune to attempts to bypass the defense [18]. We now present Canary Bit as an extension of Secure Bit 2 using a similar technique to protect non-control data from buffer overflow attacks. When data is read from an input buffer, each word is flagged with a bit that indicates it should not be trusted as a pointer address. If a buffer overflow has occurred that resulted in the overwriting of a pointer variable, the next time that pointer is dereferenced, the process will terminate with a segmentation fault.

The rest of this paper is organized as follows. In section 2, we will provide some background material on related work and offer a high-level overview of our system design. Section 3 formalizes some of the concepts discussed in the paper and provides a formal proof of the validity of the Canary Bit approach. In section 4, we will detail our implementation using the QEMU hardware emulator. Section 5 sets forth an evaluation of our system, demonstrating the correctness of the functionality. Section 6 analyzes some of the issues related to implementing Canary Bit as defined in this work. We end with sections 7 and 8 describing future work still to be performed on the system and some concluding remarks.

2 BACKGROUND

2.1 Related Work

Many researchers have explored the causes of buffer overflows and have proposed several different types of solutions [19, 20, 16]. One type of solution includes code analysis and modification [21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]. Another common group of approaches are hardware-based methods that leverage segmentation, separation of data and control memory, or various tainting tactics [18, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43]. Many techniques leverage the operating system to help enforce control [44, 45, 46, 47, 48, 49]. And a last group of tools utilize various types of cryptography and obfuscation [50, 51, 52, 53, 54, 55]. A thorough analysis of many of these systems can be found in [18] or [20].

While many of these techniques are effective, none of them are complete [18]. Some protect only against stack-based attacks or known vulnerabilities. Others can protect control data but make no effort to protect data pointers. Some that offer complete protection suffer from compatibility or performance issues. In short, no panacea exists for buffer overflow protection.

Perhaps the most similar approach to our work would be the system designed by Chen *et al.* [33]. Their approach is to mark each word of memory read from input with a bit to indicate that the word has potentially been tainted. A crucial distinction between our approach and theirs is that their mechanism can be turned off in software in order to reduce the risk of false positives or to create a work-around for backward compatibility. The problem with this approach is that it reduces the security guarantee for the system. Specifically, this creates a vector of attack that can potentially be exploited. Our approach is to make the integrity check mandatory to ensure the validity of all data pointers.

2.2 System Overview

As we built Canary Bit as an extension of the Secure Bit 2 protocol [18], a thorough understanding of the similarities and differences between the two systems will aid the reader in exploring the details of this work. Secure Bit 2 works by maintaining a parallel memory, where a single bit corresponds to a word in the system’s main memory. When data crosses a domain boundary in an application, the operating system executes an operation to switch to *sbit_write* mode. Then, when the data is written to a word of memory, the corresponding Secure Bit is set. After writing to memory, the operating system leaves *sbit_write* mode and the application continues processing. If the process later attempts to use that word as a piece of control data (e.g., as a return address, or a function pointer), the processor will check that the bit has been set and will raise an exception. The process aborts with a segmentation fault instead of jumping to a potentially hazardous memory location.

Similarly, Canary Bit maintains an additional parallel memory that also has each bit corresponding to a word in main memory. It is important to note that there is no stipulation that Canary Bit and Secure Bit memories be separate. In our implementation, we placed the start of the Canary Bit memory after the end of the Secure Bit memory. This was done for simplicity, since the address mapping for Secure Bit was already in place. However, if one is to add both mechanisms to a new system, the bits could be interleaved. Each even bit could be used as the Secure Bit, and each odd used as the Canary Bit, or vice versa. Such implementation choices are left to the system designer and not dictated by either protocol.

As in Secure Bit, the operation system switches to *sbit_write* mode when passing data across domains. However, this idea of when data crosses domain boundaries differs between the two systems. For example, consider the line of code in Figure 1. In this case, the Secure Bit for the *argv* word would be marked, and the Canary Bit would not. The reason for this is that Secure Bit does not trust any input, whereas

Canary Bit is only looking for data pointer buffer overflows. Thus, Canary Bit will only mark the input data if it is copied to another location. Otherwise, the system would raise an exception whenever the *argv* pointer gets dereferenced, preventing applications from ever being able to use input data. That is certainly not desirable. Furthermore, this is clearly not an example of a buffer overflow, so the protocol is able to handle this legitimate use of input data correctly.

```
intVar = (unsigned int)(argv[1][0]);
```

Figure 1: Passing input data to a variable

Another crucial difference between the implementation of Secure Bit and Canary Bit is that the Secure Bit is passed along to all data that is derived from the marked word. For Canary Bit, that is not the case. The intent of Canary Bit is to act similarly to a canary word mechanism (e.g., StackGuard [56]) by signaling that data has gone beyond the end of the buffer, but doing so in a manner that cannot be bypassed. Canary Bit is not intended to permanently mark input data as tainted. Given the scope of the present work is solely to protect against overflows, it is unavoidable that application designers must retain the responsibility of verifying the format of their input data.

Lastly, the timing of the checks in Secure Bit and Canary Bit differ. Intuitively, this occurs because the two systems have different purposes. Secure Bit is used to prevent the application from jumping to an invalid location, so its check occurs during a branch instruction. Canary Bit, used to preserve the integrity of data pointers, must be validated when a pointer is dereferenced in either a load or a store operation. Specifically, the check occurs during a load or store where the source address has been loaded into a general purpose register. As we will describe below, this is the way that pointer dereferences are performed. To perform the check in our implementation using the ARM architecture, we created new instructions, *ldvd* (load-and-validate) and *stvd*

(store-and-validate). When the processor encounters either of these instructions, it performs the check in parallel with the intended memory reference. If the Canary Bit has been set, the processor raises an exception and aborts the process.

2.3 Alternate Implementations

Before proceeding to the details of the present work, we shall digress by offering some notes on two additional implementation choices for Canary Bit and explain why the current implementation was favored. One option was to use a combination of the existing Secure Bit protocol with a canary word. The second option was to create a new operating system mode that is used for checking the Canary Bit. For simplicity, we shall refer to these alternatives as Canary Word and Canary Mode, respectively.

One may be tempted to dismiss the Canary Word approach based solely on the fact that researchers have shown canary word systems, such as StackGuard, can be bypassed while leaving the canary intact [13]. However, a vital part of defeating such a system was to use a buffer overflow against a data pointer. That means, assuming Canary Word is used to protect all data pointers, one must bypass the Canary Word defense mechanism in order to bypass the Canary Word. Clearly, such logic is circular, and the weakness of this approach lies elsewhere.

Indeed, the crucial flaw in the Canary Word approach is the protection of pointers on the heap. Given the unstructured nature of memory on the heap, it is impossible to compile library functions such as *malloc* in a manner that Canary Words would be placed in the correct locations. As an example, consider the code in Figure 2.

Since *malloc* only receives an integer that indicates the number of bytes to allocate, it does not have adequate information regarding the structure of the desired usage of that portion of memory to add Canary Words correctly. In this case, both of the data structures take up two words of memory, but *malloc* cannot distinguish between them. Thus, any such canary-based methods have the inherent flaw that they cannot

```
typedef struct A {
    int * ptr1;
    int * ptr2;
} A;
typedef struct B {
    int buf[2];
} B;
/* ... */
ptr = (struct A *)malloc(2 * sizeof(struct A));
```

Figure 2: Example of allocating heap memory for a struct

adequately protect pointers on the heap.

Regarding Canary Mode, there are no such explicit disadvantages to implementing such a technique. Rather, the choice relied upon a few subtle factors. First, creating the Canary Mode entails modifying the kernel appropriately. If this is done incorrectly, the results could be disastrous. Creating Canary Bit, on the other hand, primarily involved only the addition of new instructions to the GNU assembler. The structure of the code made this a straightforward task of adding one line per instruction. Any changes we made to the Linux kernel were trivial in nature and posed minimal risk of introducing a bug.

Second, adding the Canary Mode would involve creating more than just a single new mode in order to preserve the functionality of Secure Bit. Since some pointer dereferences occur during a store operation, there must be a new mode that checks the Canary Bit while *sbit_write* is enabled and another when it is not. This may appear to be a minor concern, but creating new modes of operations introduces additional complexity to the kernel. This results in an increase to the risk of introducing a bug into the kernel.

Third, there would be an unknown performance overhead resulting from switching into and out of the Canary Mode on every memory load or store. As this approach was not chosen, we have not analyzed what the actual impact would be. A quick analysis

of a trivial program¹ yielded 191 loads or stores out of a total of 333 operations. In the worst case, switching between modes on every operation could double the number of operations required. To reiterate, though, this is only intended as an intuitive analysis and may not be a true reflection of the performance impact of using the Canary Mode approach.

In summary, in order to create a complete data pointer protection mechanism while minimizing the risk and impact of the design change, the Canary Bit approach proved to be the optimal technique.

¹The program in question is one of the demonstration programs described in the evaluation section. Specifically, it is the vulnerable application that is used in the *free* attack. The source code can be found in the appendix.

3 FUNDAMENTALS

In this section, we will state our precise definition of what is intended by a buffer overflow, a buffer overflow attack, and the integrity of a memory address. We will then narrow our focus to attacks against data pointers, the necessary condition for such attacks, and simple corollary. We use the definitions offered by Piromsopa and Enbody [18], but with some alteration to Definition 2.

3.1 Definitions

We define a buffer overflow as follows:

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 memory location, one that the data was not intended to go into.

Intuitively, the term “overflow” conjures an imagine of filling something beyond its capacity. If one were to overflow a bucket with water, the result is that some of the water would pour onto the floor upon which the bucket was placed. Similarly, if a computer program uses a memory buffer and attempts to copy more data into that buffer than space allocated, some of the data will spill over into the following portion of memory. Given this definition, we can define an attack that uses a buffer overflow as follows:

Definition 2: A buffer overflow attack on a data pointer is the inducement of a buffer overflow by an external source, resulting in the modification of the value of said pointer.

It is important to emphasize that our definition of a buffer overflow attack explicitly identifies an external source as the cause of the overflow. That is, the data provided must come from an untrusted domain and cross the boundary into a trusted location. As such, it is clear that all input data must be treated with a certain amount of skepticism. However, restricting the programming model to disallow input data is obviously not a desirable goal. Instead, we simply aim to track the data to ensure that it does not overwrite any data pointers.

The results of a buffer overflow attack on a data pointer can be extensive. Access control programs may limit what data a user can read based on that user’s permissions. If that permission level is stored in memory, a buffer overflow attack on a data pointer can be used to modify that level illicitly. In section 5, we will describe a real attack on the *free* function that corrupts the structure of data in the heap and allows an attacker to copy input data of their choosing into an arbitrary memory location.

Our final definition is also that provided by Piromsopa and Enbody [18]:

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 (including input).

The logical entailment of combining Definitions 2 and 3 is that a buffer overflow attack requires violating the integrity of a memory location used to store a data pointer. The contrapositive of this statement is that preserving the integrity of the data pointer memory location is sufficient to thwart a buffer overflow attack. Then the goal of our proposed system is simply to ensure the validity of the address stored in a data pointer.

3.2 Limitation of Scope

Before proceeding farther with the formalization of our system definitions, we feel it necessary to clarify the scope of our work by highlighting two examples of what Canary Bit is not.

First, Canary Bit is not intended to provide universal protection from all buffer overflow attacks. Definition 2 above makes it clear that our work is focused on attacks against data pointers. That is, we do not intend to protect control data, such as return addresses, jump addresses, or function pointers². The preservation of control data falls under the domain of Secure Bit 2.

Second, Canary Bit does not protect against format string attacks. Figure 3 shows an example of this type of vulnerability. Format string attacks are often described in literature concerning buffer overflow attacks [4], but they do not fit into our definition of a buffer overflow attack. In fact, an attacker can perform a format string attack *without* overflowing a buffer. Clearly, such malfeasance is of a different sort than is the focus of our present work.

```
printf(buffer);  
/* should be printf("%s", buffer); */
```

Figure 3: A format string vulnerability

3.3 Formalization of Concept

We now continue with the formalization of our system description by returning to our stated goal of preserving the integrity of data pointers. We start by declaring the following invariant:

²Incidentally, our system *does* indirectly prevent attacks against function pointers. The standard approaches to create such an attack would be to use a basic overflow or to corrupt a data pointer in order to perform an arbitrary-copy attack. In the latter case, dereferencing the corrupted data pointer (which is a necessary step) would trigger the Canary Bit defensive mechanism. In the former case, both Canary Bit and Secure Bit are prepared to catch the overflow, but it is Canary Bit that would strike first. The pointer must be dereferenced first to load the location of the next instruction. The jump can only occur after this is done. Hence, Canary Bit would flag the error first.

Invariant 1: Canary Bit ensures that the contents of a data pointer have not been corrupted by an external source.

Clearly, ensuring that Invariant 1 holds preserves the integrity of the memory address, and thereby guarantees that the system cannot be vulnerable to a buffer overflow attack against data pointers. Thus, if we can demonstrate that Canary Bit does, in fact, satisfy this invariant, then we have formally proven that our system provides a robust defense against buffer overflow attacks. We will use induction as the basis of our proof.

We state as our inductive hypothesis that, if Canary Bit has successfully prevented against buffer overflow attacks on data pointers, then it will continue to do so.

We begin with the basis case of a trivial program that does nothing, and simply returns control to the calling program immediately. Clearly, such a program has no input data, so there cannot be any data from an external source. Consequently, there can be no data pointer corrupted, and, by Definition 2, there cannot be a buffer overflow attack. Furthermore, this same rationale holds for any program that never processes input data. That is, any program that never processes input data cannot be corrupted by an external source and cannot be the victim of a buffer overflow attack. Thus, Canary Bit ensures the safety of all such programs.

We begin our discussion of the inductive case by noting that any program that receives external data must have a first instruction that does so. The sequence of operations that precedes that instruction can then be viewed as a sub-program which never processed input. Our basis case proved that Canary Bit ensures the safety of this sub-program. For the instruction processing the input, then, there are two cases: Either the data is being read, or the data is being copied to a new location. It is obvious that the former cannot create a buffer overflow, as nothing is being written

into a buffer. Therefore, we only need to consider the latter case.

If the data is being copied to a new location and it is crossing a domain boundary, then the system will have switch to *sbit_write* mode, and the Canary Bit will be set for the destination word. Now, there are three possibilities to consider: First, the word is never used as a data pointer. In this case, there has been no data pointer corrupted, so the invariant still holds. Second, the word is used as a data pointer and dereferenced. The dereference operation will require the execution of either *ldvd* or *stvd*, depending on whether the pointer is used for a load or a store. These operations will check the Canary Bit and will raise an exception. Thus, Canary Bit will have successfully defended against the attack. The third case is that the application will overwrite the memory location with an immediate value. The result of this is that the Canary Bit will be cleared at that time, as immediate values are defined in the program itself and do not arrive from an external source. Therefore, Canary Bit guarantees that the copied location is never used as a data pointer, and the invariant holds for the inductive case.

Thus, as the invariant holds for both the basis and the induction, we have formally proven that Invariant 1 holds in all cases. The immediate consequence, then, is that Canary Bit guarantees the system provides a robust defense against buffer overflow attacks on data pointers.

4 IMPLEMENTATION

4.1 ARM Architecture Changes

In order to implement Canary Bit, we had to make small changes to the ARM architecture. The first change required was to claim a portion of the *current program status register* (CPSR). The ARM Architecture Reference Manual [57] specification for the bits in the CPSR are shown in Figure 13.

31	30	29	28	27	26		8	7	6	5	4	3	2	1	0
N	Z	C	V	Q	Reserved			I	F	T	M 4	M 3	M 2	M 1	M 0

Figure 4: Assigned CPSR bits

In the implementation of Secure Bit 2 [18], Piromsopa and Enbody used bit 15 to indicate that the system had switched to *sbit_write* mode. Similarly, we used bit 14 to indicate that the system is in a similar mode for setting the Canary Bit. Consequently, the bit mask 0x00008000 is used to access the Secure Bit, 0x00004000 is used for the Canary Bit, and 0x0000c000 is used for simultaneous access to both.

As we will describe in the section on the changes to the Linux kernel, our current implementation uses a single mode for writing both the Secure Bit and the Canary Bit. Thus, either both bits 14 and 15 in the CPSR are set, or both are clear. Although this may appear to be redundant, we chose this approach because future implementations of Canary Bit might use separate modes for the two bits.

The second change to the ARM architecture was to add the new instructions. Specifically, we wanted to create *ldvd*, *ldvdb*, *stvd*, and *stvdb*, such that the new instructions imitated *ldr*, *ldrb*, *str*, and *strb*, respectively³. The only difference between the existing and the new instructions is whether or not the Canary Bit of the memory

³While we describe all four new instructions in this section, we will typically refer only to *ldvd* for the sake of expediency. Unless the context specifies otherwise, descriptions of *ldvd* apply to the other new instructions, as well.

location is checked.

As the instructions were intended to be nearly identical, our goal was to create a format for the new instructions that was as close to the original as possible. Figure 5 shows the format of the load and store instructions. We will briefly describe most of the instruction format here, and refer interested readers to the full description in the ARM Architecture Reference Manual [57].

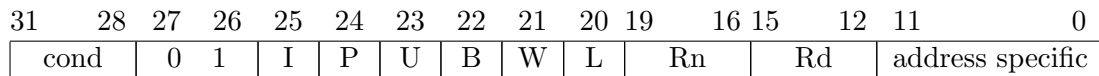


Figure 5: Format of a load/store word/unsigned byte instruction

The I, P, U, and W bits are used to determine the addressing mode. Of these bits, we are only concerned with the I bit, which indicates that bits 0–11 hold an immediate value. The B bit, if set, specifies that the operation is accessing an unsigned byte; if clear, the operation is accessing a word. Setting the L bit indicates the operation is a load, while clearing it creates a store. The Rn bits encode the number of the general purpose register that is used as the base for the memory access. The Rd bits encode the number for the “destination” register. In the case of a load, the value is stored in the register indicated by Rd, whereas for a store, register Rd holds the value that will be written to memory.

The last section of bits with which we are concerned is the condition field, bits 28–31. These bits tell the processor under what circumstances the operation can be skipped. Given that our new instructions are performing a security-related check, we do not want the processor to be able to skip the operation. Fortunately, setting all four bits for *ldr* or *str* instructions results in undefined behavior. Thus, we chose to encode *ldvd* and *stvd* in the same format as the original instructions, but setting all four of the bits in the condition field. Figure 6 shows a translation from the assembly language code to binary format for an example usage of *ldvd*.

The next section describes the changes made to the GNU assembler. We will

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
1111	0	1	1	P	1	0	W	1	0001	0000	0000	0000	0000	1000	

Figure 6: Encoding of operation `ldvd r0, [r1, #8]`

show that the decision to keep the original format for the instruction proved to be advantageous. This design choice allowed us to leverage the existing GNU code for processing the addressing mode and argument portions of the assembly language instructions.

4.2 GNU Assembler

The structure of the GNU assembler (`gas`) code facilitates the easy addition of new instructions. Within the source code directory for `binutils-2.17`, the `gas/config` subdirectory contains the files that dictate the translation of assembly language instructions to binary executable code. For the ARM architecture, the file is `gas/config/tc-arm.c`. To add the `ldvd` and `stvd` instructions, we added the lines shown in Figure 7. These lines utilize a C preprocessor macro `TUF`. The `T` indicates the presence of a thumb variant. Note that our utilities never generate the thumb variant of these instructions. So that behavior in `gas` is ignored in practice. The `UF` specifies that the ARM variant of the operation is executed unconditionally by setting the value `0xf` in the conditional field of the instruction.

<code>TUF(ldvd,</code>	<code>4100000,</code>	<code>f4100000,</code>	<code>2,</code>	<code>(RR, ADDR),</code>	<code>ldst, t_ldst),</code>
<code>TUF(ldvdb,</code>	<code>4500000,</code>	<code>f4500000,</code>	<code>2,</code>	<code>(RR, ADDR),</code>	<code>ldst, t_ldst),</code>
<code>TUF(stvd,</code>	<code>4000000,</code>	<code>f4000000,</code>	<code>2,</code>	<code>(RR, ADDR),</code>	<code>ldst, t_ldst),</code>
<code>TUF(stvdb,</code>	<code>4400000,</code>	<code>f4400000,</code>	<code>2,</code>	<code>(RR, ADDR),</code>	<code>ldst, t_ldst),</code>

Figure 7: Lines added to `gas/config/tc-arm.c`

The arguments to the `TUF` macro are (in order) (1) the instruction name, (2) the binary format for the thumb instruction, (3) the binary format for the ARM instruction, (4) the number of arguments to the operation, (5) the list of operation

arguments, (6) the gas function for processing the arguments for the ARM instruction, and (7) the gas function for processing the thumb arguments⁴.

This means that the normal version of *ldvd* will be initially encoded as 0xf4100000. Next, the *ldst* field specifies the function that processes the remaining portion of the assembly code operation. Specifically, gas expects *ldvd* instructions to take two arguments, the first is a general-purpose register (RR) and the second is a memory address (ADDR).

One important point here is that the *ldst* and *t_ldst* entries are the same that are used for translating the existing *ldr* and *str* instructions. This is due to the fact that *ldvd* and *ldr* are identical instructions from the perspective of the assembler. Their behavior only differs during the actual execution of the binary instruction.

After adding these lines, we recompiled binutils as a cross-compiler executable. For our implementation, we did not require a full re-compilation of GCC. Instead, we took the re-compiled bin/arm-none-linux-gnueabi-as and replaced our existing version. Then, when we used the gcc command, the compiler used the new assembler and correctly compiled our ARM executables.

In addition to modifying the assembler, we added to `opcodes/arm-dis.c` the lines shown in Figure 8. These lines are compiled into the disassembler (`objdump`). Similar to the process for gas, we replaced `bin/arm-none-linux-gnueabi-objdump` with the re-compiled version. As a result, we have two vital tools for compiling and inspecting ARM executables.

```
{ARM_EXT_V1, 0xf4100000, 0xfc100000, "ldvd%c%22'b%t\t%12-15r, %a"},  
{ARM_EXT_V1, 0xf4000000, 0xfc100000, "stvd%c%22'b%t\t%12-15r, %a"},
```

Figure 8: Lines added to `opcodes/arm-dis.c`

⁴In truth, fields (6) and (7) are not the names of functions themselves. Rather, the values in those fields are prepended with “do_” to specify the function name. Thus, *do_ldst* is the gas function that processes the encoding for these operations.

4.3 Perl Utility Script

The previous section detailed the steps of adding *ldvd* and *stvd* to the gas and objdump executables, which are both part of the binutils distribution. While re-compiling binutils is a straightforward process, modifying and re-compiling the entire GCC suite as a cross-compiler can be a daunting process. Consequently, to simplify the process of building small ARM applications to evaluate our implementation, we created a utility script called *cangcc* (Canary Bit GCC).

The core functionality of *cangcc* is to execute a system call that compiles any C code to assembly, replace particular calls to *ldr* with *ldvd*, then make another system call to finish the compilation process that creates the ARM executable. The most important part of the script is the *ReplaceLdr* function, shown in Figure 9.

```
sub ReplaceLdr {
    my $asm = shift;
    open my $IN, $asm or
        Error ($RC_ERROR, "Could not open $asm for reading");
    open my $OUT, ">$asm.new" or
        Error ($RC_ERROR, "Could not open $asm.new for writing");
    while (<$IN>) {
        chomp();
        if ($_ =~ m/^\s*ldrb?\s+\w+, \s+[r\d+[, \]]/) {
            my $line = $_;
            $line =~ s/^\s*(\s*)ldr/${1}ldvd/;
            print $OUT "$line\n";
        } elsif ($_ =~ m/^\s*strb?\s+\w+, \s+[r\d+[, \]]/) {
            my $line = $_;
            $line =~ s/^\s*(\s*)str/${1}stvd/;
            print $OUT "$line\n";
        } else {
            print $OUT "$_\n";
        }
    }
    close $OUT;
    close $IN;
    rename("$asm.new", "$asm");
}
```

Figure 9: The ReplaceDir function in the *cangcc* utility script

This function takes a single argument, which is the name of an assembly language file. The *while* loop reads the assembly code line-by-line, searching for *ldr* and *str* instructions, as well as their byte-counterparts, *ldrb* and *strb*. These operations take two arguments. The first is a general purpose register (e.g., *r0*, which matches the “\w+” in the regular expressions on lines 9 and 13). The second argument is an address. In the case of pointer dereferences, the target address has previously been loaded into a register. In the assembly code, that register name and an optional offset are surrounded by brackets (which matches the “[r\d+[,\\]” portion of the regular expressions⁵).

It is important to note that these regular expressions only match the register names that begin with the letter “r.” This is because GCC compiles pointer dereferences using the unbanked general purpose registers, R0–R7. In implementations where this functionality is built directly into GCC instead of using a utility script, there would be no need to restrict the *ldvd* instruction to the unbanked registers.

4.4 Linux

The only changes to the Linux kernel needed were to modify the bit masks that switch the processor into and out of *sbit_write* mode. As described previously, bits 14 and 15 in the CPSR determine whether or not the processor will set the Canary Bit and Secure Bit, respectively. Consequently, the bit mask 0x0000c000 can be used to set these bits, and 0xffff3fff can be used to clear them. To accomplish this, we changed `arch/arm/lib/sbit_macro.S` as shown in Figure 10. We made similar changes in `arch/arm/lib/copy_template_s.S`. Figure 11 shows an example.

⁵Astute readers will notice that this regular expression actually omits the closing bracket if there is an offset. Examples of the exact match include “[r0]” and “[r1,]”. In the former case, the address is stored in register *r0*. In the latter, there is a base value, stored in register *r1*, to which an offset is added. This format is common for array indices. For instance, if *r1* contains the address at the start of an int array called *iArray*, then `[r1, #4]` is used to access the memory location for `iArray[1]`.

```

.macro set_sbitmode
push    {r3}
mrs     r3, cpsr
/* MSK -- changed 0x00008000 to 0x0000c000 for cbit */
orr     r3, r3, #0x0000c000
msr     cpsr_x, r3
pop     {r3}
.endm

.macro clr_sbitmode
push    {r3}
mrs     r3, cpsr
/* MSK -- changed 0xffff7fff to 0xffff3fff for cbit */
and     r3, r3, #0xffff3fff
msr     cpsr_x, r3
pop     {r3}
.endm

```

Figure 10: The modified version of arch/arm/lib/sbit_macro.S

```

/* changed #0x8000 to #0xc000 for cbit */
msr     cpsr_x, #0xc000
str1b   r0, r3, ne, abort=21f
str1b   r0, r4, cs, abort=21f
str1b   r0, ip, cs, abort=21f
msr     cpsr_x, #0x0000

```

Figure 11: Sample of the modifications made in arch/arm/lib/copy_template.s.S

4.5 QEMU ARM Emulator

In order to test our implementation of the Canary Bit extension, we modified QEMU to act as an ARM hardware emulator. QEMU is a machine emulator that supports multiple architectures [1]. As such, the source code consists of generic wrappers that are shared by all the supported machine types, as well as target-specific functionality. The ARM-specific code is stored in the target-arm subdirectory. In this section, we will explain the changes made to both the generic QEMU code and that contained in the target-arm subdirectory.

4.5.1 QEMU ARM-specific Modifications

Each architecture that QEMU supports defines its own structure of registers. In order to support all machine types, QEMU defines its own registers, T0, T1, and T2. QEMU operates by translating target instructions into micro operations which move data into these registers, operates on them, and stores the result in the processor-specific registers⁶. Figure 12 shows the changes we made to *CPUARMState*, as defined in `target-arm/cpu.h`. The definitions of *cbit0*, *cbit1*, and *cbit2* correspond to the QEMU registers T0, T1, and T2, respectively. When a value is copied from an ARM register to a QEMU register, the Canary Bit value is copied into the matching *cbit* register. The single value *cbitF* is used to store a cache of the value of the Canary Bit stored in the CPSR to allow QEMU to operate faster. The remaining definitions represent the Canary Bit for each ARM register.

The *cbit_regs* is for all 16 of the general purpose registers. The ARM architecture contains a mode for high-speed data transfers called FIQ. These transfers use a separate copy of registers R8–R12. When the CPU switches to FIQ, the values in these registers are stored in a copy for USR mode; then the stored FIQ values are loaded. After the transfer, the values are again switched to the stored USR mode versions. Registers R13 and R14 are used for exception handling. ARM keeps six copies of these registers, one per CPU mode, in a bank. When the mode changes, the appropriate banked register is then loaded.

In addition to the changes to the structure of *CPUARMState*, we added references to the Canary Bit in *cpsr_read* and *cpsr_write*. These functions are used for accessing and setting the CPSR, such as when the CPU changes modes. As shown in Figure 13 on Page 22, bit 14 of the CPSR has been used to control whether the Canary Bit will be set for data passing across domains.

⁶It should be obvious that, as QEMU is a piece of software, the term “register” refers to a data structure in the QEMU code that represents the actual hardware register.

```

typedef struct CPUARMState {
    /* for T0, T1, T2 */
    uint8_t cbit0,cbit1,cbit2;
    /* Regs for current mode. */
    uint8_t cbit_regs[16];
    /* Banked registers. */
    uint32_t cbit_banked_r13[6]; /* Canary bit */
    uint32_t cbit_banked_r14[6]; /* Canary bit */
    /* These hold r8-r12. */
    uint32_t cbit_usr_regs[5]; /* Canary bit */
    uint32_t cbit_fiq_regs[5]; /* Canary bit */
    /* cpsr flag cache for faster execution */
    uint8_t cbitF; /* Cached cbit */
    /* ... */
}

```

Figure 12: Modifications to the CPUARMState structure

```

#define CPSR_cbit (1 << 14)
#define CACHED_CPSR_BITS \
    (CPSR_T | CPSR_Q | CPSR_NZCV | CPSR_sbit | CPSR_cbit)
/* Return the current CPSR value. */
static inline uint32_t cpsr_read(CPUARMState *env) {
    /* ... */
    return env->uncached_cpsr | (env->NZF & 0x80000000) |
        (ZF << 30) | (env->CF << 29) | ((env->VF & 0x80000000) >> 3) |
        (env->QF << 27) | (env->thumb << 5) | (env->sbitF <<15) |
        (env->cbitF << 14);
}
/* Set the CPSR. */
static inline void cpsr_write(CPUARMState *env,
                             uint32_t val, uint32_t mask) {
    /* ... */
    if (mask & CPSR_cbit)
        env->cbitF=((val & CPSR_cbit) !=0);
    mask &= ~CACHED_CPSR_BITS;
    env->uncached_cpsr = (env->uncached_cpsr & ~mask) |
        (val & mask);
}

```

Figure 13: Modifications to *cpsr_read* and *cpsr_write* in target-arm/cpu.h

Once the changes to the CPU structure were defined in target-arm/cpu.h, we modified several functions in target-arm/helper.c. These functions define the QEMU behavior for common CPU functions, such as resetting and switching modes. Figure 14 shows that resetting the CPU clears the Canary Bit for register R15, which is

also the program counter (PC).

```
void cpu_reset(CPUARMState *env) {
    /* ... */
    env->cbit_regs[15]=0;
}
```

Figure 14: Clearing the PC Canary Bit on a CPU reset

Figure 15 shows the changes made to the *switch_mode* function. If the CPU is either switching into or out of FIQ mode, then the FIQ registers (R8–R12) must be swapped. Then the exception handling registers (R13 and R14) must be swapped. Our modifications here are to swap the Canary Bits along with the register data.

```
void switch_mode(CPUState *env, int mode) {
    old_mode = env->uncached_cpsr & CPSR_M;
    /* ... */
    /* If switching to or from FIQ mode, replace registers R8-R12 */
    if (old_mode == ARM_CPU_MODE_FIQ) {
        memcpy (env->cbit_fiq_regs, env->cbit_regs + 8,
                5 * sizeof(uint32_t));
        memcpy (env->cbit_regs + 8, env->cbit_usr_regs,
                5 * sizeof(uint32_t));
        /* ... */
    } else if (mode == ARM_CPU_MODE_FIQ) {
        memcpy (env->cbit_usr_regs, env->cbit_regs + 8,
                5 * sizeof(uint32_t));
        memcpy (env->cbit_regs + 8, env->cbit_fiq_regs,
                5 * sizeof(uint32_t));
        /* ... */
    }
    /* Replace the exception handling registers */
    i = bank_number(old_mode);
    env->cbit_banked_r13[i] = env->cbit_regs[13];
    env->cbit_banked_r14[i] = env->cbit_regs[14];
    /* ... */
    i = bank_number(mode);
    env->cbit_regs[13] = env->cbit_banked_r13[i];
    env->cbit_regs[14] = env->cbit_banked_r14[i];
    /* ... */
}
```

Figure 15: Canary Bit additions to switching the CPU mode

The exception handling functions, *do_interrupt* also required modifications for the

Canary Bit handling. Since the exception handling routines are considered safe, we cleared the Canary Bit in the CPSR. The value of the PC before the interrupt gets stored in register R14. Consequently, we do the same for the Canary Bit for the PC, copying it to the Canary Bit for R14. We finish by clearing the Canary Bit for the PC. These changes are shown in Figure 16.

```

void do_interrupt(CPUARMState *env) {
    /* ... */
    env->cbitF=0;
    env->cbit_regs[14] = env->cbit_regs[15];
    env->cbit_regs[15] = 0;
}

```

Figure 16: Modifications to the QEMU interrupt handler

The last changes made to target-arm/helper.c were to *get_phys_addr*. This function translates a virtual memory address into a physical memory address. To implement Canary Bit, we had to add a reference to the local Canary Bit value to the calls to *ldl_phys*, as the prototype of that function had changed. This is shown in Figure 17.

```

static int get_phys_addr(CPUState *env, uint32_t address,
    int access_type, int is_user, uint32_t *phys_ptr, int *prot) {
    uint8_t cbit;
    /* ... */
    if (/* MMU is not disabled */) {
        /* Pagetable walk. */
        /* Lookup l1 descriptor. */
        table = (env->cp15.c2 & 0xffffc000) |
            ((address >> 18) & 0x3ffc);
        desc = ldl_phys(table, &sbit, &cbit);
        /* ... */
        if (/* Page size is not 1 MB */) {
            /* Lookup l2 entry. */
            table = (desc & 0xfffffc00) | ((address >> 10) & 0x3fc);
            desc = ldl_phys(table, &sbit, &cbit);
        }
        /* ... */
    }
}

```

Figure 17: Changes to *get_phys_addr* in target-arm/helper.c

The remainder of the files in the target-arm subdirectory concern the translation

of ARM instructions to QEMU micro operations and the behavior of those micro operations. The driver file for converting these operations is `target-arm/translate.c`. Here, we modified two functions. Figure 18 shows the first of these two, `msr_mask`. This function returns a bit mask that can be used for extracting bits from the CPSR. Our change was to add the necessary bits to extract the Canary Bit for the mode.

```
static uint32_t msr_mask(DisasContext *s, int flags, int spsr) {
    uint32_t mask = 0;
    /* ... */
    /* Mask out undefined bits. */
    mask &= 0xf90fc3ff;
    /* Mask out state bits. */
    if (!spsr)
        mask &= ~0x01000020;
    /* Mask out privileged bits. */
    if (IS_USER(s))
        mask &= 0xf80fc200;
    return mask;
}
```

Figure 18: Modification of `msr_mask` in `target-arm/translate.c`

The second function modified in `target-arm/translate.c` was `disas_arm_insn`, which retrieves the next instruction in binary format and generates the relevant ARM micro operations. As noted before, the encoding of `ldvd` and `ldr` instructions are identical, with the exception of the conditional code stored in bits 28–31. Thus, the necessary changes included copying the block for processing `ldr` nearly verbatim. The relevant portions of the function are shown in Figure 19. The first change was to add a local variable to pass to the `ldl_code` function. This function was changed in a similar manner as `ldl_phys`, which we mentioned above. Once the instruction is retrieved, the condition code is examined. QEMU continues examining the instructions bits until it reaches the line `} else if ((insn & 0x0c000000) == 0x04000000) {`. If this test passes, then QEMU has reached an `ldvd` instruction.

Within the block of code shown in Figure 19, there are calls to functions that

begin with “gen.”,⁷ such as *gen_movl_T1_reg* and *gen_op_ldvdl_user*. These functions are dynamically generated during the compilation of QEMU, as we will describe shortly. When one of these functions is executed, QEMU generates the appropriate micro operations for execution.

It is crucial to understand the contexts within which QEMU executes. QEMU operates by creating chunks of micro operations. One of these chunks is called a *TranslationBlock*. Calling *disas_arm_insn* is one of the steps of creating a *TranslationBlock*. QEMU continues translating until it receives a branch or jump operation. However, the execution of the micro operations does not occur until the entire *TranslationBlock* has been created. Consequently, QEMU cannot check the status of the Canary Bit during translation, because an earlier operation in the *TranslationBlock* might have set or cleared it. Instead, checking the Canary Bit must be done during the execution of the generated micro operations, which occurs at a later point.

In order for QEMU to translate the ARM instructions to micro operations, QEMU needs a mapping between the two. This mapping is defined by the file *target-arm/op.c*. Any function defined in that file (or any included file) will have a prototype of the form `void OPPROTO op_bx_T0(void)`. Specifically, the QEMU dynamic generator uses *OPPROTO* as a keyword and will generate, in this example, *gen_op_bx_T0*. These are the functions that are then called from *disas_arm_insn* to create the QEMU micro instructions. Figure 20 shows the first portion of changes to *target-arm/op.c*. The definitions of *CBIT0* through *CBITF* are used through the file. Defining *REGNAME*, *REG*, *sbitREG*, and *cbitREG* and including *target-arm/op_template.h* is done once for each of the 16 ARM general purpose registers.

Figure 21 shows a sample of the changes to *target-arm/op_template.h*. The C preprocessor macro *glue* combines the two arguments, substituting the values defined in *target-arm/op.c*. Consequently, combining the code in Figure 20 with that in

⁷Although it is conventional to place the comma inside the quotation marks, we deliberately placed it outside in this case to emphasize that the comma was not a part of the string described.

```

static void disas_arm_insn(CPUState * env, DisasContext *s) {
    unsigned int cond, insn, val, op1, i, shift, rm, rs, rn, rd, sh;
    uint8_t sbit, cbit;
    insn = ldl_code(s->pc, &sbit, &cbit);
    s->pc += 4;
    cond = insn >> 28;
    if (cond == 0xf){ /* block for unconditional instructions */
        if ((insn & 0x0c000000) == 0x04000000) {
            /* ldvd - format 1111 01LP UBW1 Rn-- Rd-- xxxx xxxx xxxx */
            rn = (insn >> 16) & 0xf;
            rd = (insn >> 12) & 0xf;
            gen_movl_T1_reg(s, rn); /* generate move Rn to T1 */
            i = (IS_USER(s) || (insn & 0x01200000) == 0x00200000);
            if (insn & (1 << 24)) /* is the P bit set? */
                gen_add_data_offset(s, insn);
            if (insn & (1 << 20)) { /* load */
                if (insn & (1 << 22)) {
                    if (i) gen_op_ldvdub_user();
                    else gen_op_ldvdub_kernel();
                } else {
                    if (i) gen_op_ldvdl_user();
                    else gen_op_ldvdl_kernel();
                }
                if (rd == 15) gen_bx(s);
                else gen_movl_reg_T0(s, rd);
            } else { /* store */
                gen_movl_T0_reg(s, rd);
                if (insn & (1 << 22)) {
                    /* similar to load, but gen_op_stvd* is called */
                }
            }
            if (!(insn & (1 << 24))) {
                gen_add_data_offset(s, insn);
                gen_movl_reg_T1(s, rn);
            } else if (insn & (1 << 21))
                gen_movl_reg_T1(s, rn); {
            }
            return;
        }
        /* ... */
    }
    /* ... */
}

```

Figure 19: Added functionality for translating *ldvd* in target-arm/translate.c

```

#define CBIT0 (env->cbit0)
#define CBIT1 (env->cbit1)
#define CBIT2 (env->cbit2)
#define CBITF (env->cbitF)

#define REGNAME r0
#define REG (env->regs[0])
#define sbitREG (env->sbit_regs[0])
#define cbitREG (env->cbit_regs[0])
#include "op_template.h"

```

Figure 20: Changes made to target-arm/op.c for handling loads to and from QEMU registers

Figure 21 creates the function prototypes *op_movl_T0_r0* and *op_movl_r0_T0*. The QEMU dynamic code generator then creates the functions *gen_op_movl_T0_r0* and *gen_op_movl_r0_T0*, which can be called from target-arm/translate.c. These functions are used to transfer data (including the Canary Bit) between an ARM register and a QEMU register.

```

void OPProto glue(op_movl_T0_, REGNAME)(void) {
    T0 = REG;
    SBIT0 = sbitREG;
    CBIT0 = cbitREG;
}
void OPProto glue(glue(op_movl_, REGNAME), _T0)(void) {
    SET_REG (T0);
    sbitREG = SBIT0;
    cbitREG = CBIT0;
}

```

Figure 21: Examples of the changes made to target-arm/op_template.h

After constructing every combination of instruction for moving data between the ARM and QEMU registers, target-arm/op.c defines the functionality of the remaining micro operations. One pertinent group of operations is shown in Figure 22. When moving an immediate value to a QEMU register, we want to clear the Canary Bit of that register, as that data is not taken from user input. Conversely, if data is transferred from one QEMU register to another, the Canary Bit should also be transferred. Table 1 lists the micro operations in which the Canary Bit of a QEMU register is

cleared or replaced⁸. Note that QEMU uses the convention of listing the destination register name first in the micro operation name. E.g., *op_movl_T0_T1* copies the value from T1 into T0.

```

void OPProto op_movl_T0_0(void) {
    T0 = 0;
    SBIT0 = 0;
    CBIT0 = 0;
}
void OPProto op_movl_T0_im(void) {
    T0 = PARAM1;
    SBIT0 = 0;
    CBIT0 = 0;
}
void OPProto op_movl_T0_T1(void) {
    T0 = T1;
    SBIT0 = SBIT1;
    CBIT0 = CBIT1;
}

```

Figure 22: Moving immediate values to, or swapping values of QEMU registers

Operation	Canary Bit Result
op_movl_T0_0	T0 cleared
op_movl_T0_im	T0 cleared
op_movl_T0_T1	T0 replaced with T1
op_movl_T1_im	T1 cleared
op_movl_T2_im	T2 cleared
op_movl_T0_cpsr	T0 cleared
op_movl_T0_spsr	T0 cleared
op_shrl_T1_0	T1 cleared
op_shrl_T1_0_cc	T1 cleared
op_shrl_T2_0	T2 cleared
op_clz_T0	T0 cleared
op_movl_T2_T0	T2 replaced with T0
op_movl_T0_T2	T0 replaced with T2

Table 1: Summary of QEMU micro operations and the result for the register’s Canary Bit

The last portions of code from target-arm/op.c that we modified for Canary

⁸This excludes the moves between QEMU registers and ARM registers, as well as the memory access operations.

Bit were the functions *op_movl_T0_user* and *op_movl_user_T0*. Figure 23 shows the changes to the former. The latter is omitted, as it is identical, except the assignment statements are reversed. These functions allow the CPU to access user mode registers from privileged modes.

```

void OPPROTO op_movl_T0_user(void) {
    int regno = PARAM1;
    if (regno == 13) {
        T0 = env->banked_r13[0];
        SBIT0=env->sbit_banked_r13[0];
        CBIT0=env->cbit_banked_r13[0];
    } else if (regno == 14) {
        T0 = env->banked_r14[0];
        SBIT0=env->sbit_banked_r14[0];
        CBIT0=env->cbit_banked_r14[0];
    } else if ((env->uncached_cpsr & 0x1f) == ARM_CPU_MODE_FIQ) {
        T0 = env->usr_regs[regno - 8];
        SBIT0=env->sbit_usr_regs[regno - 8];
        CBIT0=env->cbit_usr_regs[regno - 8];
    } else {
        T0 = env->regs[regno];
        SBIT0=env->sbit_regs[regno];
        CBIT0=env->cbit_regs[regno];
    }
    FORCE_RET();
}

```

Figure 23: QEMU micro operation to access user registers

The final modifications we made to the ARM-specific code in target-arm were to the target-arm/op_mem.h and target-arm/op_mem_raw.h files. Figure 24 shows the code modified in the former. The first two C preprocessor macros (*MEM_LD_OP* and *MEM_ST_OP*) were the existing portions, which we modified to add the Canary Bit handling. We added the *MEM_LDVD_OP* and *MEM_STVD_OP* macros to create the appropriate micro operation code for the new *ldvd* and *stvd* ARM instructions. The difference in functionality between the old and the new macros is that the new macros produce code to check the Canary Bit of register T1, which contains the address of memory to access with either a load or store. Table 2 summarizes the

functions generated by the macros in target-arm/op_mem.h.

```
#define MEM_LD_OP(name) \
void OPCODE glue(op_ld##name, MEM_SUFFIX)(void) { \
    uint8_t tmp, ctmp; \
    T0 = glue(ld##name, MEM_SUFFIX)(T1, &tmp, &ctmp); \
    SBIT0=tmp; CBIT0=ctmp; \
    FORCE_RET(); \
}
#define MEM_ST_OP(name) \
void OPCODE glue(op_st##name, MEM_SUFFIX)(void) { \
    uint8_t tmp, ctmp; \
    tmp = SBIT0; ctmp = CBIT0; \
    if (SBITF) tmp=0xFF; \
    if (CBITF) ctmp=0xFF; \
    glue(st##name, MEM_SUFFIX)(T1, T0, tmp, ctmp); \
    FORCE_RET(); \
}
#define MEM_LDVD_OP(name) \
void OPCODE glue(op_ldvd##name, MEM_SUFFIX)(void) { \
    uint8_t tmp, ctmp; \
    if (CBIT1) { \
        T1 = 0x00000000; \
        T0 = glue(ld##name, MEM_SUFFIX)(T1, &tmp, &ctmp); \
        raise_exception(EXCP_DATA_ABORT); \
    } \
    T0 = glue(ld##name, MEM_SUFFIX)(T1, &tmp, &ctmp); \
    SBIT0=tmp; CBIT0=ctmp; \
    FORCE_RET(); \
}
#define MEM_STVD_OP(name) \
void OPCODE glue(op_stvd##name, MEM_SUFFIX)(void) { \
    uint8_t tmp, ctmp; \
    tmp = SBIT0; ctmp = CBIT0; \
    if (SBITF) tmp=0xFF; \
    if (CBITF) ctmp=0xFF; \
    if (CBIT1) { \
        T1 = 0x00000000; \
        T0 = glue(ldl, MEM_SUFFIX)(T1, &tmp, &ctmp); \
        raise_exception(EXCP_DATA_ABORT); \
    } \
    glue(st##name, MEM_SUFFIX)(T1, T0, tmp, ctmp); \
    FORCE_RET(); \
}
```

Figure 24: Canary Bit functionality added to target-arm/op_mem.h

Macro	Argument	User Mode	Kernel Mode
MEM_LD_OP	ub	gen_op_ldub_user	gen_op_ldub_kernel
	sb	gen_op_ldsb_user	gen_op_ldsb_kernel
	uw	gen_op_lduw_user	gen_op_lduw_kernel
	sw	gen_op_ldsw_user	gen_op_ldsw_kernel
	l	gen_op_ldl_user	gen_op_ldl_kernel
MEM_LDVD_OP	ub	gen_op_ldvdub_user	gen_op_ldvdub_kernel
	sb	gen_op_ldvdsb_user	gen_op_ldvdsb_kernel
	uw	gen_op_ldvduw_user	gen_op_ldvduw_kernel
	sw	gen_op_ldvdsw_user	gen_op_ldvdsw_kernel
	l	gen_op_ldvdl_user	gen_op_ldvdl_kernel
MEM_ST_OP	b	gen_op_stb_user	gen_op_stb_kernel
	w	gen_op_stw_user	gen_op_stw_kernel
	l	gen_op_stl_user	gen_op_stl_kernel
MEM_STVD_OP	b	gen_op_stvdb_user	gen_op_stvdb_kernel
	w	gen_op_stvdw_user	gen_op_stvdw_kernel
	l	gen_op_stvdl_user	gen_op_stvdl_kernel

Table 2: Micro operations generated by macros in target-arm/op_mem.h

The changes we made to target-arm/op_mem_raw.h were similar to those made to target-arm/op_mem.h. One important difference, though is that the load and store functions in target-arm/op_mem.h call lower-level functions to do the actual load or store. The functions in target-arm/op_mem_raw.h do not call these lower-level functions and perform the load or store directly. As such, the target-arm/op_mem_raw.h load and store functions call *ld_cbit_base* and *st_cbit_base*, which are used to interface with the physical memory locations that contain the Canary Bit data. Figure 25 shows the modified code in target-arm/op_mem_raw.h, and Table 3 summarizes the functions generated by the macros in target-arm/op_mem_raw.h.

Before closing the discussion of the changes made to files in target-arm, we are obliged to note functionality that we did not add. Readers who are familiar with the implementation of Secure Bit 2 [18] will observe that we did not add Canary Bit processing to data processing micro operations. This includes such operations as *op_addl_T1_T2*, which shown in Figure 26. The Secure Bit protocol assumes that all

```

#define MEM_LD_OP(name, size) \
void OPPROTO glue(op_ld##name, MEMSUFFIX)(void) { \
    TO = glue(ld##name, MEMSUFFIX)(T1); \
    SBIT0=ld_sbit_base((uint8_t *) (long)T1, size); \
    CBIT0=ld_cbit_base((uint8_t *) (long)T1, size); \
    FORCE_RET(); \
}

#define MEM_ST_OP(name, size) \
void OPPROTO glue(op_st##name, MEMSUFFIX)(void) { \
    uint8_t tmp, ctmp; tmp=SBIT0; ctmp=CBIT0; \
    if (SBITF) tmp=0xFF; \
    if (CBITF) ctmp=0xFF; \
    glue(st##name, MEMSUFFIX)(T1, TO); \
    st_sbit_base((uint8_t *) (long)T1, tmp, size); \
    st_cbit_base((uint8_t *) (long)T1, ctmp, size); \
    FORCE_RET(); \
}

#define MEM_SWP_OP(name, lname, size) \
void OPPROTO glue(op_swp##name, MEMSUFFIX)(void) { \
    uint32_t tmp; uint8_t sbit_tmp, cbit_tmp; \
    cpu_lock(); \
    tmp = glue(ld##lname, MEMSUFFIX)(T1); \
    sbit_tmp=ld_sbit_base((uint8_t *) (long)T1, size); \
    cbit_tmp=ld_cbit_base((uint8_t *) (long)T1, size); \
    glue(st##name, MEMSUFFIX)(T1, TO); \
    st_sbit_base((uint8_t *) (long)T1, SBIT0, size); \
    st_cbit_base((uint8_t *) (long)T1, CBIT0, size); \
    TO = tmp; SBIT0= sbit_tmp; CBIT0= cbit_tmp; \
    cpu_unlock(); FORCE_RET(); \
}

#define VFP_MEM_OP(p, w, size) \
void OPPROTO glue(op_vfp_ld##p, MEMSUFFIX)(void) { \
    FTO##p = glue(ldf##w, MEMSUFFIX)(T1); FORCE_RET(); \
} \
void OPPROTO glue(op_vfp_st##p, MEMSUFFIX)(void) { \
    uint8_t tmp, ctmp; \
    if (SBITF) tmp=0xFF; \
    if (CBITF) ctmp=0xFF; \
    glue(stf##w, MEMSUFFIX)(T1, FTO##p); \
    st_sbit_base((uint8_t *) (long)T1, tmp, size); \
    st_cbit_base((uint8_t *) (long)T1, ctmp, size); \
    FORCE_RET(); \
}

```

Figure 25: Modifications made to target-arm/op_mem_raw.h

Macro	Arguments	Raw Mode Operation
MEM_LD_OP	ub, 1	gen_op_ldub_raw
	sb, 1	gen_op_ldsb_raw
	uw, 2	gen_op_lduw_raw
	sw, 2	gen_op_ldsw_raw
	l, 4	gen_op_ldl_raw
MEM_ST_OP	b, 1	gen_op_stb_raw
	w, 2	gen_op_stw_raw
	l, 4	gen_op_stl_raw
MEM_SWP_OP	b, ub, 1	gen_op_swp_raw
	l, l, 4	gen_op_swp_raw
VFP_MEM_OP	s, l, 4	gen_op_vfp_lds_raw
	d, q, 8	gen_op_vfp_ldd_raw
	s, l, 4	gen_op_vfp_sts_raw
	d, q, 8	gen_op_vfp_std_raw

Table 3: Micro operations generated by macros in target-arm/op_mem_raw.h

```

void OPProto op_addl_T1_T2(void) {
    T1 += T2;
    SBIT1=SBIT_ADD(SBIT1,SBIT2);
}

```

Figure 26: Existing functionality of *op_addl_T1_T2* is preserved

data derived from user input is untrusted and should be marked as such. However, since the purpose of Canary Bit is solely to detect buffer overflows into data pointers, we do not want to pass the Canary Bit on to derived data. Consequently, these types of functions should leave the Canary Bit of the QEMU registers unmodified.

4.5.2 QEMU Generic Code Modifications

The first change to the generic QEMU code that we changed was to allocate space for the Canary Bit memory emulation. This was done in the file vl.c, which defines the *main* function for QEMU. Figure 27 shows the allocation of memory for the Canary Bit space. In our current implementation, we created the Canary Bit memory to be the same size as the main memory. To simplify the access to the Canary Bit for purposes of demonstration, we used a full byte to connote what would be a single

Canary Bit in a real implementation.

```
phys_cbit_size = phys_ram_size;
phys_cbit_base = qemu_vmalloc(phys_cbit_size);
if (!phys_cbit_base) {
    fprintf(stderr, "Could not allocate physical memory for cbit\n");
    exit(1);
}
```

Figure 27: Allocation of space for the Canary Bit memory in vl.c

After allocating the memory containing the Canary Bit data, we created the functions *ld_cbit*, *st_cbit*, *ld_cbit_base*, and *st_cbit_base* to access the Canary Bit memory. The two “_base” functions translate the main memory address to the corresponding Canary Bit location. Then the other two functions are used to actually read or set the Canary Bit in memory. One thing to note is that the value stored in the Canary Bit is either 0 or 0xff, as our implementation uses bytes for the Canary Bit. When reading the Canary Bit, if the value stored is non-zero, then the result returned will be 0xff. Similarly, when storing the Canary Bit, if the data is not zero, then the Canary Bit value stored is 0xff. These functions are shown in Figures 28, 29, 30, and 31.

```
static inline uint8_t ld_cbit(void *ptr, int size) {
    int i;
    uint8_t* myptr;
    uint8_t val=0;
    myptr=ptr;
    for (i=0;i<size;i++)
        val|=*(myptr+i);
    if (val!=0) val=0xff;
    return val;
}
```

Figure 28: The *ld_cbit* function defined in cpu-all.h

Additional changes that we made to *cpu-all.h* include adding the In addition, to the changes listed above, we added the Canary Bit parameter *cbit_buf* to the function prototypes of several functions. Table 4 lists the prototypes modified.

```

static inline void st_cbit(void *ptr, uint8_t val, int size) {
    int i;
    uint8_t* myptr;
    myptr=ptr;
    for (i=0;i<size;i++)
        *(myptr+i)=(val!=0)?0xff:0;
}

```

Figure 29: The *st_cbit* function defined in *cpu-all.h*

```

static inline uint8_t ld_cbit_base(void *ptr, int size) {
    uint8_t *myptr;
    myptr=(uint8_t *) ptr - phys_ram_base + phys_cbit_base;
    return ld_cbit(myptr, size);
}

```

Figure 30: The *ld_cbit_base* function defined in *cpu-all.h*

```

static inline void st_cbit_base(void *ptr, uint8_t val, int size) {
    uint8_t *myptr;
    myptr=(uint8_t *) ptr - phys_ram_base + phys_cbit_base;
    st_cbit(myptr,val,size);
}

```

Figure 31: The *st_cbit_base* function defined in *cpu-all.h*

ldub_phys	stb_phys
lduw_phys	stw_phys
ldl_phys	stl_phys
ldq_phys	stq_phys
cpu_memory_rw_debug	stl_phys_notdirty
cpu_physical_memory_write	cpu_physical_memory_read
cpu_physical_memory_rw	

Table 4: Function prototypes modified in *cpu-all.h*

The next group of functions we modified were the functions that are responsible for reading and writing memory. The first of these is *cpu_physical_memory_rw*, which is defined in the file *exec.c*. Figure 32 shows the outline of the function, as well as the highlights related to Canary Bit. This function is notable because it is the only instance of the generic QEMU functionality in which the Canary Bit and Secure Bit are handled differently. This function is responsible for reading and writing data from a physical device. As a result, the Secure Bit must be marked for such reads and

writes. The Canary Bit, however, must not be marked because the data has not yet been put into a memory buffer through a function such as *strcpy*. There are several functions that act as wrappers for *cpu_physical_memory_rw*, and they are listed in Table 5.

It is critical to emphasize that if we were to set the Canary Bit in this circumstance, applications would be prevented to ever read input data. Consequently, the Canary Bit cannot be set here. Instead, we set the bit later using the software-controlled memory management functions defined in *softmmu_header.h* and *softmmu_template*, and discussed below. This different treatment highlights an important distinction between Canary Bit and Secure Bit. While Secure Bit shows equal distrust to all input data, Canary Bit only shows distrust to data written to a memory buffer.

<code>ldub_phys</code>	<code>stb_phys</code>
<code>lduw_phys</code>	<code>stw_phys</code>
<code>get_phys_addr_code</code>	<code>stq_phys</code>

Table 5: Functions that reference *cpu_physical_memory_rw*

While the previous functions define the memory accesses used by physical devices, the changes described above concerning the files in the *target-arm* subdirectory use other low-level functions. Figures 33 and 34 shows the general outline of two of these functions that are defined in *exec.c*. The *ldq_phys* and *stl_phys_notdirty* functions have the same structure.

The last group of load and store functions are generated by the *softmmu_header.h* and *softmmu_template.h* files. These files are used by the code in *target-arm* to generate the ARM memory access operations. As the code in these files are obtuse C preprocessor macros, including the full source code here would do little to enlighten the reader to the details of Canary Bit. Instead, Figures 35 and 36 show the logical structure of the code as it relates to Canary Bit.

The final changes to the QEMU generic code were made to the debugging code in

```

void cpu_physical_memory_rw(target_phys_addr_t addr, uint8_t *buf,
    int len, int is_write, uint8_t *sbit_buf, uint8_t *cbit_buf) {
    /* ... */
    if (is_write) {
        if ((pd & ~TARGET_PAGE_MASK) != IO_MEM_RAM) {
            /* Read from the buffer and write it to IO
             * This is trusted, so the Canary Bit is not set
             * ... */
        } else {
            /* Read from the buffer and copy it to a memory location.
             * Set the Secure Bit, but not the Canary Bit
             * ... */
            sbit_ptr = phys_sbit_base + addr1;
            cbit_ptr = phys_cbit_base + addr1;
            if (sbit_buf != NULL) memcpy(sbit_ptr, sbit_buf, 1);
            else st_sbit(sbit_ptr, 0, 1);
            st_cbit(cbit_ptr, 0, 1);
        }
    } else {
        if ((pd & ~TARGET_PAGE_MASK) > IO_MEM_ROM && !(pd & IO_MEM_ROMD)) {
            /* Reading from an IO buffer is trusted,
             * so don't trust the Canary Bit
             * ... */
            if (cbit_buf != NULL) stl_p(cbit_buf, 0);
            /* similar for stw_p and stb_p, depending on input size */
            * ... */
        } else {
            /* Reading from memory and copying into a buffer,
             * set the Canary Bit accordingly
             * ... */
            if (cbit_buf != NULL) {
                cbit_ptr = phys_cbit_base + (pd & TARGET_PAGE_MASK) +
                    (addr & ~TARGET_PAGE_MASK);
                memcpy(cbit_buf, 0, 1);
            }
        }
    }
    /* ... */
    if (cbit_buf != NULL) cbit_buf += 1;
}

```

Figure 32: The *cpu_physical_memory_rw* function in *exec.c*

```

uint32_t ldl_phys(target_phys_addr_t addr, uint8_t *sbit, uint8_t *cbit) {
    if (/* read from IO, which is trusted */) {
        /* ... */
        cbit_val=0;
    } else { /* read from RAM, so carry the Canary Bit */
        /* ... */
        cbit_ptr = phys_cbit_base + (pd & TARGET_PAGE_MASK) +
            (addr & ~TARGET_PAGE_MASK);
        cbit_val = ld_cbit(cbit_ptr, 4);
    }
}

```

Figure 33: The *ldl_phys* function defined in *exec.c*

```

void stl_phys(target_phys_addr_t addr, uint32_t val, uint8_t sbit,
             uint8_t cbit) {
    if (/* write to IO, so ignore Canary Bit */) {
        /* ... */
    } else { /* write to memory, so write to the Canary Bit */) {
        /* ... */
        cbit_ptr = phys_cbit_base + (pd & TARGET_PAGE_MASK) +
            (addr & ~TARGET_PAGE_MASK);
        st_cbit(cbit_ptr, 0, 4);
    }
}

```

Figure 34: The *stl_phys* function defined in *exec.c*

```

if (/* IO access */) {
    cbit_res=0;
} else {
    cbit_res = ld_cbit_base((uint8_t *) (long) physaddr, DATA_SIZE);
}
*cbit= cbit_res;
return res;

```

Figure 35: Sample logical flow for a Canary Bit load in *softmmu.header.h* and *softmmu.template.h*

disas.c, *monitor.c*, and *gdbstub.c*. The modified functions call *cpu_physical_memory_rw*. Consequently, we added the Canary Bit parameter to these calls. The modified functions are listed in Table 6.

```
if (/* Write to IO */) {
    /* Ignore Canary Bit */
} else {
    physaddr = addr + env->tlb_table[is_user][index].addend;
    st_cbit_base((uint8_t*) physaddr, cbit, DATA_SIZE);
}
```

Figure 36: Sample logical flow for a Canary Bit store in softmmu_header.h and softmmu_template.h

target_read_memory
cpu_memory_rw_debug
gdb_handle_packet
memory_dump
do_sum

Table 6: Modified debug functions in disas.c, monitor.c, and gdbstub.c

5 EVALUATION

In order to evaluate our Canary Bit implementation, we constructed test cases that represented different types of buffer overflow attacks. Specifically, we tested attacks against vulnerable buffers adjacent to data pointers located on the stack and the heap. Then, to illustrate a more interesting and realistic attack, we created an attack that emulates the heap corruption attack using a buffer overflow vulnerability in the *free* C library function. We close this section with a demonstration that our Canary Bit implementation does not interfere with valid uses of input data.

5.1 Sample Attack Wrapper Code

Launching a buffer overflow attack requires leveraging an attack vector to push the constructed attack code into a vulnerable application buffer. Our method for demonstrating the success of our implementation was to create a wrapper program that calls the vulnerable application, passing the attack data to the victim through a command line argument. Figure 37 shows a sample format for the attacking source code⁹. The char array *buf* contains the attack data, where the last four bytes make up the portion that overflows the victim's buffer. Note that in all of our attacks, we overflow the buffer with a memory address. As ARM is a big-endian architecture, the bytes of the address are in reverse order. In this sample code, we are passing memory address 0x00010920 to the vulnerable application. Figure 38 shows a sample main function that we used for the vulnerable applications.

⁹The convention is that the vulnerable code (without the Canary Bit) was compiled as *ldr*, while the Canary Bit-enabled version was compiled as *ldvd*. Then, the wrapper code was compiled as *wldr* and *wldvd*, depending on which version of the vulnerable code it will call.

```

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    char buf[8];
    buf[0] = 'a'; buf[1] = 'b'; buf[2] = 'c'; buf[3] = 'd';
    buf[4] = 0x20; buf[5] = 0x09; buf[6] = 0x01; buf[7] = 0x00;
    char **arr = (char **)malloc(sizeof(char *)*4);
    /* arr[0] is the executable name --
     * ./ldr is for the non-Canary Bit version
     * ./ldvd is for the Canary Bit version*/
    arr[0] = "./ldr";
    arr[1] = buf;
    arr[2] = '\0';
    printf("In wrapper, arr[1] = %s\n", arr[1]);
    printf("Calling execv\n");
    execv(arr[0], arr);
}

```

Figure 37: A sample buffer overflow attack program

```

int main (int argc, char *argv[]) {
    int i;
    for (i=0; i<argc; i++)
        printf("argv[%d] @%p=%s\n", i, &argv[i], argv[i]);
    if (i<1) {
        printf("Please enter 1 argument"); return -1;
    }
    printf("Sample program.\n");
    vulnerable(argv);
    printf("Program exits normally.\n");
}

```

Figure 38: The main method for a vulnerable application

5.2 Stack Smashing Attacks

The most common and well-studied type of buffer overflow attack is the stack-smashing attack. Figure 39 shows a vulnerable function where *strcpy* is used to write into the buffer, *buf*. The overflow occurs and overwrites the address stored in pointer *b*. In this code, *x* and *y* are global variables that are referenced by the pointers on the stack.

Our attack input consists of an eight-character string, whose last four characters encode the address 0x00010920, the address of the global variable *x*. Figure 40 shows

```

int x = 5; int y = 10;
int vulnerable(char **argv) {
    int * a, * b;
    char buf[4];
    buf[0] = '\0'; buf[1] = '\0'; buf[2] = '\0'; buf[3] = '\0';
    a = &x; b = &y;
    printf("Before overflow, b = 0x%08x\n",b);
    printf(" *b = %d\n",*b);
    strcpy(buf,argv[1]);    // buffer overflow
    printf("After overflow, b = 0x%08x\n",b);
    printf(" *b = %d\n",*b);
}

```

Figure 39: A function vulnerable to a stack-based buffer overflow

```

# ./wldr
In wrapper, arr[1] = abcd [non-printable characters]
Calling execv
argv[0] @0xbef79eb4=./ldr
argv[1] @0xbef79eb8=abcd [non-printable characters]
Sample program.
Before overflow, b = 0x00010924
 *b = 10
Before overflow, b = 0x00010920
 *b = 5
Program exits normally.

```

Figure 40: Overflowing a stack variable without Canary Bit

the results of executing the attack on the application without the Canary Bit *ldvd* instructions. Figure 41 shows the results using *ldvd*. The segmentation fault indicates that the attack was successfully trapped.

```

# ./wldvd
In wrapper, arr[1] = abcd [non-printable characters]
Calling execv
argv[0] @0xbea2beb4=./ldvd
argv[1] @0xbea2beb8=abcd [non-printable characters]
Sample program.
Before overflow, b = 0x00010924
 *b = 10
Before overflow, b = 0x00010920
Segmentation fault

```

Figure 41: Canary Bit successfully stopping a stack-smashing attack

5.3 Unstructured Heap Attacks

Though heap-based attacks are not as common as stack-smashing¹⁰, they are not impossible. Figure 42 shows an example of a program that is vulnerable to a heap attack. For easy reference, the structure is nearly identical to the stack-smashing example, and the *main* function is the same as in Figure 39.

```
int x = 5; int y = 10;
int vulnerable(char **argv) {
    int i;
    int ** a, ** b;
    char * buf = (char*)(malloc(12 * sizeof(char)));
    for (i = 0; i < 12; i++) buf[i] = '\0';
    a = (int**)(buf + 8); *a = &x;
    b = (int**)(buf + 4); *b = &y;
    printf("Address of x 0x%08x\n",&x);
    printf("Before overflow, *b 0x%08x,\n", *b);
    printf(" **a = %d; **b = %d\n",**a,**b);
    strcpy(buf,argv[1]);    // buffer overflow
    printf("After overflow, *b 0x%08x,\n", *b);
    printf(" **a = %d; **b = %d\n",**a,**b);
}
```

Figure 42: An application vulnerable to a heap-based buffer overflow

In this example, our attack string encodes the address 0x00010bc4, which is the address of the global variable *x* once again. Figure 43 shows the results of executing the attack on the application without the Canary Bit *ldvd* instructions, and the attack again succeeds. Figure 44 shows the results using *ldvd*. As before, the segmentation fault indicates that the attack was successfully trapped.

5.4 Heap Corruption Using *free*

Attacking the *free* C library function requires an understanding of the memory allocation structure for the heap. To demonstrate this attack, we created our own memory structure to ease the illustration for the reader. The full source code for this attack

¹⁰This is most likely due to the lack of a return address to corrupt, as well as a more unpredictable structure in general.

```
# ./wldr
In wrapper, arr[1] = abcd [non-printable characters]
Calling execv
argv[0] @0xbefb3eb4=./ldr
argv[1] @0xbefb3eb8=abcd [non-printable characters]
Sample program.
Before overflow, *b = 0x00010bc8,
  **a = 5; **b = 10
After overflow, *b = 0x00010bc4,
  **a = 5; **b = 5
Program exits normally.
```

Figure 43: Overflowing a heap variable without Canary Bit

```
# ./wldvd
In wrapper, arr[1] = abcd [non-printable characters]
Calling execv
argv[0] @0xbeabfeb4=./ldvd
argv[1] @0xbeabfeb8=abcd [non-printable characters]
Sample program.
Before overflow, *b = 0x00010bc8,
  **a = 5; **b = 10
After overflow, *b = 0x00010bc4,
Segmentation fault
```

Figure 44: Canary Bit successfully stopping a heap-based overflow

is included in the appendix due to its length. In this section, we will only include the portions immediately relevant to the explanation.

First, we created a structure whose definition is shown in Figure 45. The *mblock* structures are used to create a doubly-linked list, as shown in Figure 46. As memory allocations are handled, the heap gets divided into chunks of varying sizes, each of which is separated by an *mblock*. To handle a new request, the pointers are traversed to find a chunk that is large enough to supply the number of bytes. Once such a block is found, it is divided into two chunks with a new *mblock* structure between them. The pointers in the preceding and following *mblocks* are then modified to include this new structure.

The process for freeing a chunk of memory works similarly. The *free* function traverses the linked list until it finds the *mblock* for the memory to be freed. If this

```

typedef struct mblock {
    struct mblock * prev;
    struct mblock * next;
    unsigned int alloc : 1;
    unsigned int rest : 31;
} mblock;

```

Figure 45: The structure of a memory block for allocating heap chunks

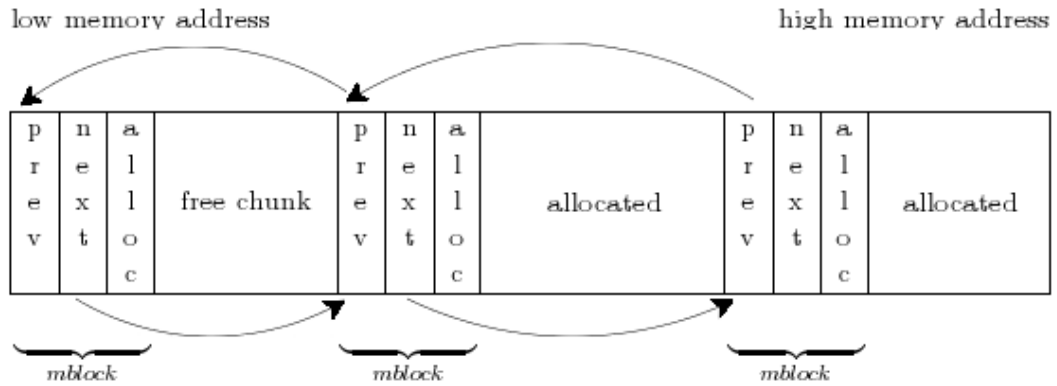


Figure 46: A visual representation of the memory allocation structure

chunk is preceded by one that is already free, the two are merged. This is done by updating the *next* pointer of the preceding *mblock* and the *prev* pointer of the following *mblock*. Figure 47 shows the replacement of the pointer values.

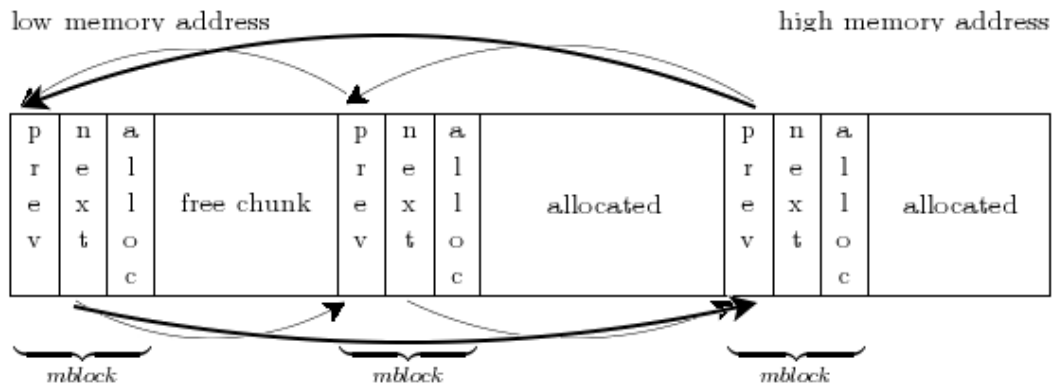


Figure 47: Replacing pointer values when a chunk is freed

This pointer replacement serves as the crux needed to use *free* to perform an arbitrary

rary copy buffer overflow attack. The attack is performed by writing data beyond the end of an allocated heap chunk, corrupting the pointer values stored in the following *mblock*. Figure 48 shows the code for our simulated version of *free*. It is the last line `mptr->prev->next = mptr->next;` that is manipulated during the attack. The first word overwritten is the value stored in `mptr->prev`. In our attack, we set this value to be the the word preceding our intended target. The second word overwritten is `mptr->next`. Thus, this value will got stored in our target address.

```

void myFree (void * p) {
    struct mblock * mptr = (struct mblock *)StartOfHeap;
    if ((char*)p < StartOfHeap || (char*)p >= EndOfHeap) return;
    while (mptr->next < (struct mblock *)p) mptr = mptr->next;
    mptr->alloc = 0;
    if (mptr->prev < mptr && mptr->prev->alloc == 0) {
        mptr->prev->next = mptr->next;
        if (mptr->next < (struct mblock *)EndOfHeap)
            mptr->next->prev = mptr->prev;
    }
    mptr = mptr->next;
    if (mptr < (struct mblock *)EndOfHeap && mptr->alloc == 0) {
        if (mptr->next < (struct mblock *)EndOfHeap &&
            mptr->next >= (struct mblock *)StartOfHeap)
            mptr->next->prev = mptr->prev;
        /* The next line is manipulated in the attack */
        mptr->prev->next = mptr->next;
    }
}

```

Figure 48: Our simulated *free* function

Figure 49 shows the code for the vulnerable function that we used to launch our attack. The *strcpy* overwrote the data stored in the following *mblock*. Then, that chunk of memory was freed, ensuring that the last call to *myFree* will merge the two chunks, performing the arbitrary copy.

Finally, Figure 50 shows the *main* function for our vulnerable application. Not shown in these figures¹¹ is the *isValid* global variable, which is located at memory location 0x00010c5c. This is the value that we used to overwrite the `mptr->prev`

¹¹Thorough readers can see the full code in the appendix.

```
void vulnerable(char **argv) {
    char * a, * b, * c, *d;
    printf("In vulnerable\n");
    a = (char*)myMalloc(8);
    b = (char*)myMalloc(8);
    c = (char*)myMalloc(20);
    d = (char*)myMalloc(20);
    myFree(c);
    strcpy(b,argv[1]);
    myFree(b);
    myFree(a);
}
```

Figure 49: The function referencing the vulnerable *free*

value. To simplify our demonstration, we designed our vulnerable system so that the *checkAccess* function only checks to see if the value of *isValid* is zero. Thus, our attack only needed to write a non-zero value to our target. Since *mptr->next* contained an address (*i.e.*, it was non-zero), there was no need to overwrite this value.

The result was that our attack wrote the address stored in *mptr->next* into the *isValid* variable, which caused the second call to *checkAccess* to produce a different result. Specifically, we successfully managed to change a global variable that was used as an access control. Obviously, this kind of a vulnerability in a production system could give an attacker unfettered access to sensitive information. Figure 51 shows the result without the Canary Bit protection, and Figure 52 shows that Canary Bit successfully stopped this attack.

5.5 Using Valid Input Data

We conclude this section by showing that our implementation permits for legitimate access to valid input data. That is, Canary Bit allows applications to use valid input data without flagging it as an error. Figure 53 shows some sample code that we used to test that Canary Bit allows valid accesses to input data.

Figure 54 shows the result of executing the previous code without the Canary Bit

```

void checkAccess() {
    if (isValid == 0) printf("User does not have access\n");
    else printf("User DOES have valid access\n");
}
int main (int argc, char *argv[]) {
    initHeap();
    if (argc < 1) {
        printf("Please enter 1 argument");
        return -1;
    }
    printf("Sample program.\n");
    checkAccess();
    vulnerable(argv);
    checkAccess();
    printf("Program exits normally.\n");
}

```

Figure 50: The wrapper functions for the program vulnerable to the *free* attack

```

# ./wldr
In wrapper, arr[1] = aaaaaaaa\[non-printable characters]
Sample program.
User does not have access
In vulnerable
User DOES have valid access
Program exits normally.

```

Figure 51: Attacking *free* without Canary Bit

```

# ./wldvd
In wrapper, arr[1] = aaaaaaaa\[non-printable characters]
Sample program.
User does not have access
In vulnerable
Segmentation fault

```

Figure 52: Canary Bit successfully stopping an attack on *free*

by providing the string “1” as input. The *notvulnerable* function copies the value from *argv* into the local variable, which it then uses as an integer value to offset an existing pointer. Figure 55 shows the result of the same test, but with Canary Bit enabled. As expected, the behavior is identical, with the implication that Canary Bit allows legitimate access to input data.

Having shown the code, we would be remiss to neglect mentioning a false posi-

```

int x = 5; int y = 10;
int notvulnerable(char **argv) {
    char * c;
    char d;
    char buf[4];
    buf[0] = 'a'; buf[1] = 'b'; buf[2] = 'c'; buf[3] = 'd';
    c = buf; d = argv[1][0];
    printf("Before adding the offset, c (0x%08x) -> %c, d = %c\n",c,*c,d);
    if (d < '0' || d > '3') {
        printf("Illegal input: %s\n",argv[1]);
    } else {
        d -= '0';
        c += d;
        printf("After adding the offset, c (0x%08x) -> %c\n",c,*c);
    }
}

```

Figure 53: Canary Bit permits valid access to legitimate input data

```

# ./wldr
In wrapper, arr[1] = 1
Calling execv
argv[0] @0xbeee8eb4=./ldr
argv[1] @0xbeee8eb8=1
Sample program.
Before adding the offset, c (0xbeee8d24) -> a, d = 1
After adding the offset, c (0xbeee8d25) -> b
Program exits normally.

```

Figure 54: Demonstrating valid input data without Canary Bit

```

# ./wldvd
In wrapper, arr[1] = 1
Calling execv
argv[0] @0xbe907eb4=./ldvd
argv[1] @0xbe907eb8=1
Sample program.
Before adding the offset, c (0xbe907d24) -> a, d = 1
After adding the offset, c (0xbe907d25) -> b
Program exits normally.

```

Figure 55: Canary Bit does not create a false positive for valid input data

tive avoided here. Many users would combine lines 12 and 13 into a single line as `c += (d - '0');`. When either the separate lines or the single line are compiled, GCC puts the value of `d` into one register and the value of `c` into another. However,

when the two values are added, the sum is written into different registers. Figure 56 shows the different compilations that occur for the two variations of the same code.

```
/* Compiling d -= '0'; c += d; */
ldrb    r3, [fp, #-13]
sub     r3, r3, #48
strb    r3, [fp, #-13]
ldrb    r3, [fp, #-13] @ zero_extendqisi2
mov     r2, r3
ldr     r3, [fp, #-20]
add     r3, r3, r2

/* Compiling c += (d - '0') */
ldrb    r3, [fp, #-13] @ zero_extendqisi2
mov     r2, r3
ldr     r3, [fp, #-20]
add     r3, r2, r3
```

Figure 56: Different GCC compilations of similar functionality

It is interesting to observe the subtlety introduced by the difference of interpretation. When the lines are written separately, the last addition can be read as, “Add the value of d to the register holding the value of c .” *I.e.*, the value of c can be considered the “base” value and the value of d is something added to it. When the lines are combined, the interpretation is reversed. The value of c is added to the base register storing the result of the subtraction. As addition is reflexive, this distinction should not matter. It is simply curious (and problematic for us) that GCC produces these different results. As a result, we were forced to choose between accepting the risk of this false positive or changing the micro operations so that anything other than a load or store operation would clear the Canary Bit. Given the goal described by our Canary Bit Invariant is to guarantee that the pointer has not been corrupted, we chose to accept the false positive. One could modify the compiler to develop a consistent translation of arithmetic like this, programmers could be required to work around this issue, or one could create a preprocessor macro to identify and correct the problem. Though this is less than ideal, we find it more acceptable than allowing

the accidental corruption of a data pointer.

6 ISSUES

In this section, we will analyze the issues and costs related to implementing Canary Bit as previously described. We will show that this technique offers a great number of benefits while limiting the cost of deployment.

6.1 Virtual Memory

Virtual memory mechanisms allow pages of memory to be swapped into and out of a backing store to provide the illusion of a large and fast memory space. It is intuitive that the Canary Bit should be swapped in and out of main memory, as well. Thus, implementing Canary Bit requires a modification to the virtual memory management routines.

The difficulty of this modification lies in the inequitable size of the swap. For example, each 4-KB page (*i.e.*, 1024 words, or 32,768 bits) of main memory has 1024 associated Canary Bits. The system designer must then make a choice of whether to copy only the 128 bytes of data holding the Canary Bits to the backing store, or to swap out a 4-KB portion of the Canary Bit memory. There are tradeoffs involved with either option.

If one chooses to copy only the relevant 128 bytes of Canary Bit data, then this introduces complexity to the paging routine, as it must perform a mapping to identify the Canary Bit space that relates to the main memory portion. The advantage of this approach, though, is that it keeps the paging mechanism synchronized. There is no need for an additional page table, because the presence of the Canary Bit can be determined directly by the presence of the associated word in main memory. Additionally, if the system is designed so that there is a separate backing store and a dedicated bus connecting it to the Canary Bit memory, the transfer could actually be done synchronously, introducing virtually no performance overhead to the swapping

routine. If there is no separate backing store, then the overhead involves the transfer of just 128 bytes of data, which adds a time cost of approximately 3%.

The other option, swapping 4-KB of Canary Bit data at a time requires the paging routine to maintain a separate page table mapping for the Canary Bit. When a page fault occurs, the routine must perform an additional check to determine if the page containing the associated Canary Bit is present. If the Canary Bit is not present, then both pages must be loaded into memory, creating a 100% increase in transfer time. However, the principle of locality dictates that this occurrence would be rare. Specifically, for 16-MB of main memory, assuming every page is referenced and the Canary Bit page is never swapped out, there would be only a single Canary Bit page fault compared with 4096 memory page faults. This produces an amortized transfer time increase of only 0.02%. In addition, this approach preserves a single page size, which could facilitate easier location of the data in the backing store by minimizing fragmentation.

From a security perspective, we note that the presence of virtual memory does not introduce a vulnerability, as the paging mechanism is entirely hidden from the user. As the user has no vector of attack, it is impossible to corrupt the Canary Bit data in transfer through a software process.

6.2 Memory Interface

Since Canary Bit introduces a new bit that must be transferred between memory and the CPU, it is obvious that there must be a way to send this bit along a data bus. The required interface for Canary Bit would be similar to that for Secure Bit 2. The memory chip would require an additional pin devoted to the Canary Bit. Similarly, an additional bit would be required on the processor. However, the authors of Secure Bit 2 point out that existing Intel-based motherboards use algorithms that do not utilize all available transfer bits [18]. Thus, these remaining bits could be used for

the Canary Bit transfer. The details of this implementation are left to the system designer.

6.3 Restriction on the Programming Model

In the previous section, we pointed out a false positive that could result from the interpretation that GCC uses while translating a problematic line of code. In Figure 57, we show the two versions of this code. The first version causes a false positive, while the other does not. As we discussed previously, this problem could be fixed with either a preprocessor macro or a modification of the compiler. As an immediate fix, though, we recommend that programmers keep the operations separate so as to avoid this issue. While this sort of mandate is not ideal, it successfully avoids the problem without sacrificing the security of the Canary Bit protocol.

```
/* version that creates a false positive */
c += (d - '0');

/* version that avoids the false positive */
d -= '0';
c += d;
```

Figure 57: Programming work-around for a false positive

6.4 Possible Attacks

Analyzing the design of proposed security mechanisms necessarily entails exploring the possible attacks. Setting or clearing the Canary Bit is wholly transparent to the programmer, as it is handled by the operating system. The direct consequence of this is that there is no way for an application developer to “turn off” the mechanism, and there is no way for an external attacker to manipulate a program to circumvent the Canary Bit check. Instead, the only possible attack would be for the application developer intentionally to use valid input data as an address to dereference. This

clearly violates our definition of a buffer overflow attack. Consequently, the Canary Bit system has been shown to be fully robust against buffer overflow attacks.

6.5 Cost Analysis

Designers of any proposed scheme must also address the costs of implementation. Here, we will describe the costs of Canary Bit and show that they are minimal compared to alternative protections.

6.5.1 Backward Compatibility

Our implementation of Canary Bit offers full backward compatibility with existing binaries, with the exception that those binaries are not granted protection from buffer overflow attacks against data pointers. That is, these programs will continue to run in their current form on Canary Bit hardware. However, in order to guarantee complete integrity of data pointers, existing code would require a re-compilation. At this time, we have not integrated the new *ldvd* instructions into the full GCC compiler, though. Instead, one would have to compile the application into an assembly language format, then use our utility script to replace the *ldr* instructions as appropriate and continue the compilation. While this may be burdensome at this point, we note that future work on the Canary Bit protocol includes integration of this translation directly into GCC.

6.5.2 Space

The most obvious cost of a Canary Bit implementation is the addition of the memory hardware. For every 1-MB of main memory a system has, one would have to add 32-KB for the relevant Canary Bits. Thus, the cost increase in terms of space is only a 3% overhead. However, this is a one-time cost, as compared to alternative schemes that add a performance cost with every process run.

6.5.3 Performance

A common element of security discussions is that of the trade-off. That is, it is assumed that any scheme will induce a penalty; it is simply a question of how that penalty occurs. In Canary Bit, that penalty was address previously, as the one-time cost of the additional hardware. The only performance reduction that can happen is the increased overhead during the page swapping routine, which was addressed previously. As for the Canary Bit operations themselves, the address integrity validation is in parallel with the memory reference, thus creating no performance penalty during operation.

7 FUTURE WORK

The most pressing need for future work done to the Canary Bit scheme would be to incorporate the replacement of *ldr* with *ldvd* instructions into GCC. As the GCC code is very complex, this is by no means a trivial task. The benefit of doing this, though, is that we could then re-compile large applications, such as the Linux kernel, the Apache server, or the Gnome desktop environment, for further empirical evaluation of our implementation.

An additional change to GCC would be to create a consistent interpretation of the false positive code described in previous sections. Establishing this behavior would eliminate the issue of false positives and would obviate the need for the restriction on the programming model.

8 CONCLUSION

In this work, we have shown an implementation of Canary Bit for the ARM architecture, using the QEMU emulator. Our implementation successfully defended against both heap-based and stack-smashing buffer overflow attacks that would overwrite data pointers. We also demonstrated that our system allows applications to use properly constructed user input in an appropriate manner. We showed that the only false positive occurs as a result of the compiler operation, and a minimal restriction on the programming model can work around this issue. We explored the costs and issues related to implementing Canary Bit and have shown these costs to be minimal compared to other systems while offering greater protection. Thus, Canary Bit offers a valuable service by protection data pointers while incurring minimal cost.

APPENDIX

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define roundword(num) ((num) & 0xffffffffc)
typedef struct mblock {
    struct mblock * prev;
    struct mblock * next;
    unsigned int alloc : 1;
    unsigned int rest : 31;
} mblock;
char * StartOfHeap; char * EndOfHeap; int isValid = 0;
void initHeap() {
    struct mblock * mptr;
    StartOfHeap = (char*)malloc(100 * sizeof(int));
    mptr = (struct mblock *)StartOfHeap;
    mptr->prev = mptr;
    mptr->alloc = 0;
    EndOfHeap = StartOfHeap + (100 * sizeof(int));
    mptr->next = (struct mblock *)EndOfHeap;
}
void * myMalloc(int size) {
    struct mblock * mptr = (struct mblock *)StartOfHeap;
    if (size < 0) return NULL;
    roundword(size);
    size += (2*(roundword(sizeof(struct mblock))));
    while ( ( (char*)(mptr->next)-(char*)mptr ) < size ||
            mptr->alloc == 1 ) {
        if ((char*)(mptr->next) < EndOfHeap) {
            mptr = mptr->next;
        } else {
            mptr = NULL; break;
        }
    }
    if (mptr == NULL) {
        return mptr;
    } else {
        mptr->alloc = (unsigned)1;
        mptr = (struct mblock *)((char*)mptr + size -
            roundword(sizeof(struct mblock)));
        mptr->prev = (struct mblock *)((char*)mptr - size +
            roundword(sizeof(struct mblock)));
        mptr->next = mptr->prev->next;
        mptr->next->prev = mptr;
        mptr->prev->next = mptr;
        mptr->alloc = 0;
        return (char*)(mptr->prev) + roundword(sizeof(struct mblock));
    }
}
}

```

```

void myFree (void * p) {
    struct mblock * mptr = (struct mblock *)StartOfHeap;
    if ((char*)p < StartOfHeap || (char*)p >= EndOfHeap) return;
    while (mptr->next < (struct mblock *)p) mptr = mptr->next;
    mptr->alloc = 0;
    if (mptr->prev < mptr && mptr->prev->alloc == 0) {
        mptr->prev->next = mptr->next;
        if (mptr->next < (struct mblock *)EndOfHeap)
            mptr->next->prev = mptr->prev;
    }
    mptr = mptr->next;
    if (mptr < (struct mblock *)EndOfHeap && mptr->alloc == 0) {
        if (mptr->next < (struct mblock *)EndOfHeap &&
            mptr->next >= (struct mblock *)StartOfHeap)
            mptr->next->prev = mptr->prev;
        mptr->prev->next = mptr->next;
    }
}

int vulnerable(char **argv) {
    char * a, * b, * c, *d;
    printf("In vulnerable\n");
    a = (char*)myMalloc(8);
    b = (char*)myMalloc(8);
    c = (char*)myMalloc(20);
    d = (char*)myMalloc(20);
    myFree(c);
    strcpy(b,argv[1]);
    myFree(b);
    myFree(a);
}

void checkAccess() {
    if (isValid == 0) printf("User does not have access\n");
    else printf("User DOES have valid access\n");
}

int main (int argc,char *argv[]) {
    initHeap();
    if (argc<1) {
        printf("Please enter 1 argument");
        return -1;
    }
    printf("Sample program.\n");
    checkAccess();
    vulnerable(argv);
    checkAccess();
    printf("Program exits normally.\n");
}

```

BIBLIOGRAPHY

- [1] Bellard, F. 2005. "QEMU, a Fast and Portable Dynamic Translator." In *Proceedings of the USENIX Annual Technical Conference*.
- [2] One, A. 1996. "Smashing the Stack for Fun and Profit." *Phrack Magazine*.
- [3] Schmidt, C. and Darby, T. 2001. "The What, Why, and How of the 1988 Internet Worm." <http://www.snowplow.org/tom/worm/worm.html>.
- [4] Chien, E. and Ször, P. 2002. "Blended Attacks Exploits, Vulnerabilities, and Buffer-Overflow Techniques in Computer Viruses." In *Proceedings of Virus Bulletin Conference*.
- [5] CERT. 2002. "CERT Advisory CA-2002-27 Apache/mod_ssl Worm." <http://www.cert.org/advisories/CA-2002-27.html>.
- [6] Microsoft. 2004. "Microsoft Security Bulletin MS04-028: Buffer Overrun in JPEG Processing (GDI+)." <http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx>.
- [7] Dean, D., Felten, E. W., and Wallach, D. S. 1996. "Java Security: From HotJava to Netscape and Beyond." In *Proceedings of the IEEE Symposium on Security and Privacy*.
- [8] U.S. Department of Energy Computer Incident Advisory Capability (CIAC). 2004. "O-130: Perl and ActivePerl win32_stat Buffer Overflow." <http://www.ciac.org/ciac/bulletin/o-130.shtml>.
- [9] Anonymous. 2001. "Once Upon a free()." *Phrack Magazine*.
- [10] Brenner, B. 2007. "Microsoft Investigates Possible New Office Flaw." http://searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci1260760,00.html.
- [11] Mimoso, M. S. 2007. "CA Backup Bug Exploitable on Vista." http://searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci1242436,00.html.
- [12] Zetter, K. 2007. "Scan This Guy's E-Passport and Watch Your System Crash." *Wired.com*.
- [13] Bulba and Kil3r. 2000. "Bypassing StackGuard and StackShield." *Phrack Magazine*.

- [14] Piromsopa, K. and Enbody, R. J. 2006. “Defeating Buffer-Overflow Prevention Hardware.” In *Proceedings of the Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking*.
- [15] Piromsopa, K. and Enbody, R. J. 2006. “Arbitrary Copy: Bypassing Buffer-Overflow Protections.” In *Proceedings of the Sixth IEEE International Conference on Electro/Information Technology*.
- [16] Pincus, J. and Baker, B. 2004. “Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns.” pp. 20–27.
- [17] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. 2005. “Non-Control-Data Attacks Are Realistic Threats.” In *Proceedings of the USENIX Security Symposium*.
- [18] Piromsopa, K. and Enbody, R. J. 2005. “Secure Bit2: Transparent, Hardware Buffer-Overflow Protection.” Tech. Rep. MSU-CSE-05-9, Michigan State University.
- [19] Cowan, C., Wagle, P., Pu, C., Beattie, S., and Walpole, J. 2000. “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade.” In *DARPA Information Survivability Conference and Expo (DISCEX)*.
- [20] Piromsopa, K. and Enbody, R. J. 2006. “Buffer-Overflow Protection: The Theory.” In *Proceedings of the Sixth IEEE International Conference on Electro/Information Technology*.
- [21] Etoh, H. 2000. “GCC Extension for Protecting Applications from Stack-Smashing Attacks.” <http://www.trl.ibm.com/projects/security/ssp/>.
- [22] Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. 2001. “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities.” In *Proceedings of the USENIX Security Symposium*.
- [23] Necula, G. C., McPeak, S., and Weimer, W. 2002. “CCured: Type-Safe Retrotting of Legacy Code.” In *Proceedings of the Principles of Programming Languages*.
- [24] Evans, D. and Larochelle, D. 2002. “Improving Security Using Extensible Lightweight Static Analysis.” In *IEEE Software*.
- [25] Xie, Y. and Aiken, A. 2006. “Static Detection of Security Vulnerabilities in Scripting Languages.” In *Proceedings of the USENIX Security Symposium*.
- [26] Pistoia, M., Banerjee, A., and Naumann, D. A. 2007. “Beyond Stack Inspection: A Unified Access-Control and Information-Flow Security Model.” In *Proceedings of the IEEE Symposium on Security and Privacy*.

- [27] Wang, X., Pan, C.-C., Liu, P., and Zhu, S. 2006. "SigFree: A Signature-free Buffer Overflow Attack Blocker." In *Proceedings of the USENIX Security Symposium*.
- [28] Hsu, F.-H. and Chieuh, T.-C. 2001. "RAD: A Compile-Time Solution to Buffer Overflow Attacks." In *International Conference on Distributed Computing Systems (ICDCS)*.
- [29] Haugh, E. and Bishop, M. 2003. "Testing C Programs for Buffer Overflow Vulnerabilities." In *Proceedings of the Symposium on Networked and Distributed System Security (NDSS)*.
- [30] Viega, J., Bloch, J., Kohno, Y., and McGraw, G. 2000. "ITS4: A Static Vulnerability Scanner for C and C++ Code." In *Proceedings of Computer Security Application Conference*.
- [31] Wilander, J. and Kamkar, M. 2002. "A Comparison of Publicly Available Tools for Static Intrusion Detection." In *Nordic Workshop on Secure IT Systems*.
- [32] Newsome, J. and Song, D. 2005. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software." In *Proceedings of the Symposium on Networked and Distributed System Security (NDSS)*.
- [33] Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., and Iyer, R. K. 2005. "Defeating Memory Corruption Attacks via Pointer Taintedness Detection." In *Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN)*.
- [34] Crandall, J. R. and Chong, F. T. 2004. "Minos: Control Data Attack Prevention Orthogonal to Memory Model." In *37th International Symposium on Microarchitecture*.
- [35] Cowan, C., Beattie, S., Johansen, J., and Wagle, P. 2003. "PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities." In *Proceedings of the USENIX Security Symposium*.
- [36] McGregor, J. P., Karig, D. K., Shi, Z., and Lee, R. B. 2003. "A Processor Architecture Defense against Buffer Overflow Attacks." In *Proceedings of the IEEE International Conference on Information Technology: Research and Education (ITRE)*.
- [37] Özdoğanoglu, H., Vijaykumar, T. N., Brodley, C. E., Kuperman, B. A., and Jalote, A. 2003. "SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address." Tech. Rep. TR-ECE 03-13, Department of Electrical and Computer Engineering, Purdue University.
- [38] PaX. 2003. "The PaX Project." <http://pax.grsecurity.net/docs/>.

- [39] Shao, Z., Zhuge, Q., He, Y., and Sha, E. H.-M. 2004. "Defending Embedded Systems Against Buffer Overow via Hardware/Software." In *Proceeding of the Computer Security Applications Conference*.
- [40] Xu, J., Kalbarczyk, Z., Patel, S., and Iyer, R. K. 2002. "Architecture Support for Defending Against Buffer Overow Attacks." In *Workshop on Evaluating and Architecting Systems for Dependability*.
- [41] Ye, D. and Kaeli, D. R. 2005. "A Reliable Return Address Stack: Microarchitectural Features to Defeat Stack Smashing." In *ACM SIGARCH*.
- [42] Inoue, K. 2005. "Energy-Security Tradeoff in a Secure Cache Architecture Against Buffer Overflow Attacks." In *ACM SIGARCH*.
- [43] Intel. "Intel Trusted Execution Technology." <http://www.intel.com/technology/security/>.
- [44] Baratloo, A., Singh, N., and Tsai, T. 2000. "Transparent Run-Time Defense Against Stack Smashing Attacks." In *Proceedings of the USENIX Annual Technical Conference*.
- [45] Riley, R., Jiang, X., and Xu, D. 2007. "An Architectural Approach to Preventing Code Injection Attacks." In *Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN)*.
- [46] Frantzen, M. and Shuey, M. 2000. "StackGhost: Hardware Facilitated Stack Protection." In *Proceedings of the USENIX Security Symposium*.
- [47] Chang, F., Itzkovitz, A., and Karamcheti, V. 2000. "User-level Resource-constrained Sandboxing." In *Proceedings of USENIX Windows Systems Symposium*.
- [48] Peterson, D. S., Bishop, M., and Pandey, R. 2002. "A Flexible Containment Mechanism for Executing Untrusted Code." In *Proceedings of the USENIX Security Symposium*.
- [49] Suh, G. E., Lee, J., and Devadas, S. 2004. "Secure Program Execution via Dynamic Information Flow Tracking." In *Architectural Support for Programming Languages and Operating Systems*.
- [50] Bhatkar, S., DuVarney, D. C., and Sekar, R. 2003. "Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits." In *Proceedings of the USENIX Security Symposium*.
- [51] Kc, G. S., Keromytis, A. D., and Prevelakis, V. 2003. "Countering Code-Injection Attacks With Instruction-Set Randomization." In *Proceedings of the 10th ACM Conference on Computer and Communication Security*.

- [52] Tuck, N., Calder, B., and Varghese, G. 2004. "Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overow." In *37th International Symposium on Microarchitecture*.
- [53] Milenkovic, M., Milenkovic, A., and Jovanov, E. 2005. "Using Instruction Block Signatures to Counter Code Injection Attacks." In *ACM SIGARCH*.
- [54] Kgil, T., Falk, L., and Mudge, T. 2005. "ChipLock: Support for Secure Microarchitectures." In *ACM SIGARCH*.
- [55] Safford, D. 2004. "The Need for TCPA."
- [56] Cowan, C., Pu, C., Maier, D., Hinton, H., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. 1998. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." In *Proceedings of the USENIX Security Symposium*.
- [57] ARM, *The ARM Architecture Reference Manual*. 2007. <http://www.arm.com/documentation/>.