A software approach for stack memory protection based on duplication and randomisation

Sahel Alouneh*

German Jordanian University,
Amman Madaba Street,
P.O. Box 35247,
Amman 11180, Jordan
Email: sahel.alouneh@gju.edu.jo
*Corresponding author

Mazen Kharbutli and Rana AlQurem

Jordan University of Science and Technology,
P.O. Box 3030,
Irbid 22110, Jordan
Email: kharbutli@just.edu.jo
Email: rkalkarem@just.edu.jo

Abstract: With software systems continuously growing in size and complexity, the number and variety of security vulnerabilities in those systems is increasing in an alarming rate. Unfortunately, all previously proposed solutions that deal with this problem suffer from shortcomings and therefore highlighting the need for further research in this vital area. In this paper, a software-based solution for stack-based vulnerabilities and attacks is proposed, implemented, and tested. The basic idea of our approach is to implement a patch tool that makes multiple copies of the return addresses in the stack, and then randomises the location of all copies in addition to their number. All duplicate copies are updated and checked in parallel such that any mismatch between any of these copies would indicate a possible attack attempt and would trigger an exception. The results of our implementation show high protection against integer overflow and buffer overflow attacks.

Keywords: stack-based protection; buffer overflow; security.

Reference to this paper should be made as follows: Alouneh, S., Kharbutli, M. and AlQurem, R. (2016) ‘A software approach for stack memory protection based on duplication and randomisation’, Int. J. Internet Technology and Secured Transactions, Vol. 6, No. 4, pp.324–348.

Biographical notes: Sahel Alouneh is currently the Dean of Faculty of Electrical Engineering and Information Technology at the German Jordanian University. He received his BSc in Electrical and Computer Engineering from Jordan University of Science and Technology, Jordan in 2000. He obtained MSc and PhD from Concordia University. Also, he worked as the Director of the Computer Centre at GJU. His research interest includes computer networks fault tolerance, security, software engineering and computer hardware design.
Mazen Kharbutli is an Associate Professor in the Department of Computer Engineering at the Jordan University of Science and Technology (JUST). He received his PhD in Computer Engineering from North Carolina State University in the USA in 2005. He received his MS in Electrical and Computer Engineering from the University of Maryland in the USA in 2002 and BS in the same field from JUST in 2000. He has several publications in computer architecture and computer security. His research interests include: high performance computer architecture, memory systems, computer security, parallel processing, and hardware support for computer security and reliability. While a PhD student, he worked under the supervision of Dr. Yan Solihin. He is a member of the Honor Society of Phi Kappa Phi. He is also a member of the IEEE.

Rana AlQurem received her MSc in Computer Engineering in 2011 from Jordan University of Science and Technology. She obtained her BSc from the Jordan University of Science and Technology. Her research interests include software security, testing, and distributed systems.

1 Introduction

Nowadays, security is one of the major concerns for computer users. Computer security threats are caused mainly by attacks that originate in the form of computer viruses, spyware, malware, etc.

Buffer overflow in the program’s stack is a very common vulnerable place for attacks to be applicable (Park and Lee, 2004). The main objective of the attacker is usually to change the control flow of the program, allowing the attacker to execute arbitrary code such as opening a new shell with the same access rights to the system as of that process attacked. If the process has a root access, so will the attacker in the new shell, leaving the whole system open for any kind of manipulation (Wilander and Kamkar, 2003).

A generic buffer overflow attack involves exploiting an unsafe code copy to overwrite the return address of a function in the stack with the address of a piece of malicious code, which is injected by the attacker and most likely also resides in the stack; when the function returns (which pops the return address from the stack), program control is transferred to the injected malicious code (Prasad and Chiueh, 2003).

Buffer overflow attacks make up approximately half of all software security vulnerabilities (Cowan et al., 2000). In the last decade, software vulnerabilities have alarmingly grown in number and variety. According to statistics from the Computer Emergency Response Team Coordination Center at Carnegie Mellon University (CERT), the number of reported vulnerabilities in software has increased by a factor of about 10x in the period between year 2000 and 2015 (CERT) as shown in Figure 1. Such vulnerabilities can appear in very popular software applications such as microsoft windows and internet explore.

Buffer overflow attacks are easily performed as evident by their widespread compared to other forms of attacks. Since most of stack-based buffer overflow attacks lead to a compromised return address, our proposed solution focused on protecting the stack from attacks against the return addresses.
Figure 1  Software vulnerabilities reported to CERT (see online version for colours)

Many approaches have been proposed to protect against stack-based attacks, especially to prevent compromised return address. The diversity of all previous proposed mechanism can be categorised into two main different solutions: software-based (Chiveh and Hsu, 2001; Vendicator; Gadaleta et al., 2009; Cowan et al., 1998; Tyagi and Lee, 2000; Pyo and Lee, 2002; Jones and Kelly, 1997; Baratloo et al., 1999; Baratloo et al., 2000; Solar Designer; Nashimoto et al., 2016) and hardware-based (Francillon et al., 2009; Xu et al., 2002; Alexander, 2005; CERT; Giasson, 2001; Eichin and Rochlis, 1989; Danger et al., 2016). Software solutions focus on existing source code while Hardware solutions need major changes in the computer and memory hardware architecture. Unfortunately, none of them has general protection from all types of possible attacks and they all suffer from serious shortcomings. First, some proposed solutions are designed specifically with many assumptions on how the attack will be executed making them suitable only for a limited range of attacks, leaving the application vulnerable to other types of attacks. In addition, such solutions are very easy to be violated in the future from attacks that are designed specifically to overcome those proposed mechanisms. Second, other proposed solutions have been shown to be vulnerable to attacks themselves, resulting in more vulnerabilities distributed in programs and lead to new types of attacks. Another shortcoming of some approaches is the need to change the source code which may be difficult and inapplicable in some cases such as for legacy code and commercial software. On the other side, some solutions require hardware changes which make them unattractive solutions.

In this paper, we proposed, implemented, and tested a new solution for stack-based attacks that provides general protection for return addresses in the stack. The proposed solution enables dynamic detection and prevention of all known return-address stack-based attacks. The idea behind our proposed solution is to generate a random number of copies of the return addresses on the stack. Then, randomise the location of all copies and locate a known stack area for each location. When the return address would be saved in the stack during running the program, it would be stored also in parallel in all random locations. Then, at a time of using this return address in the active program, all duplicate copies of that return address would be updated and checked in parallel such that any mismatch between any of these copies would indicate a possible attack attempt and would trigger an exception. The input to our software patch is a binary file
A software approach for stack memory protection based on duplication

(i.e., executable file) and the output is a more secure binary file that is more resilient to stack-based buffer overflow attacks.

This paper is organised as follows: Section 2 presents a literature review for solutions that are related to our work and also compare them with our proposed solution. Section 3 discusses the theory of stack buffer overflow and the attack methods against the return-address. Our proposed solution and the implementation are discussed in Section 4. The simulation results and evaluation of our software solution is presented in Section 5. Finally, the discussion in Section 6 summarises and concludes the paper.

2 Related work

In this section, we present and discuss the main proposed approaches that protects against stack-based attacks, and especially in preventing compromising the return address. The defense techniques are categorised into two main types: software-based or hardware-based solutions.

2.1 Software-based solutions

Software defense mechanisms are widely used and more attractive for computer users since it is mostly independent from the operating system and it does not require adding new hardware or changing in the architecture. Most of available software defense mechanisms that protect the stack against buffer overflow attacks are implemented as compiler extensions. Therefore, they require the source code of the vulnerable programs so they can be re-compiled. This requirement is very difficult in some cases especially with legacy applications and commercial software.

Return address defender (RAD) (Chiveh and Hsu, 2001) is a dynamic detection mechanism that automatically creates a repository safe area to save a copy of return addresses to protect the program against buffer overflow attacks. Also, it adds extra codes to program during its compilation to generate more secure binary code that should be compatible with existing libraries and other object files. RAD proposed two similar protection modes: MineZone RAD and Read-Only RAR. Read-Only RAR is more secure than MineZone RAD because it protects return address from the direct return address modification attacks. In contrast, Read-Only RAR is less efficient than MineZone RAD because of the updating requirement to RAR area when adding the protection instructions in the function prologues.

Stack Shield (Vendicator) is a security development tool that protects programs against stack smashing attacks at compile time without making any changes to source code. It is designed to support the GCC under Linux operating system. Stack Shield store a copy of return address in the function prologues to a previously allocated area on the beginning of data segment. Then, before the function returns, the stored value of return address is compared to the current return address. If there is a different between the two values, the program will be terminated.

In Gadaleta et al. (2009), the authors implement a countermeasure using virtualisation to protect stack against buffer overflow attacks. The proposed countermeasure relies on adding extra instructions to the architecture that is simulated by Xen hypervisor. Two instructions: callx and retx, are added to the instruction set of the virtualised processor which will be simulated to save and restore the return address respectively. The idea of
Instruction-level countermeasure is similar to RAD, but the implementation is different. The similarity comes from using mprotect() system call to allocate a protected memory to store the return addresses. Instruction-level countermeasure use two new hardware instructions in their implementation to protect the function’s return address since only these two instructions can access the previously allocated read-only memory.

The main drawback of this strategy is the environment dependency. The countermeasures can be implemented by simulating new hardware instructions only for the virtualisation technology of Xen and it cannot be used in a realistic deployment setting (Gadaleta et al., 2009). Micro and macro benchmarks show that programs protected by the countermeasures mechanism incur high overhead reaches to 30x slow-down primarily affected by the context switch needed to simulate the hardware instructions to save and restore the return address from the allocated protected memory (Gadaleta et al., 2009). The simplest factor that makes our solution much better than instruction-level countermeasures is that our patch is operating system and hardware independent. Other factor is that the overhead of output programs of our patch is much lower than the protected programs introduced by instruction-level countermeasures.

Binary rewriting defense mechanism (Prasad and Chiueh, 2003) have been proposed employing static binary translation based on the RAD solution (Chiveh and Hsu, 2001). Binary-rewriting RAD was the first approach that implements static binary translation to secure available binaries against buffer overflow attacks without requiring the existence of their source code (Prasad and Chiueh, 2003).

Since binary-rewriting add protection instruction for only the functions that access the stack for local variables, it protects stack against buffer overflow attacks only and so, it is vulnerable to the integer overflow attacks that overwrite the return address without requiring local variables allocated on stack. This could never be happen with our mechanism since we take care of all user functions in the input binary file. Moreover, binary-rewriting inherent all the limitations of RAD mechanism and nonetheless, it does not provide the same level of protection as the compiler-based mechanism.

In Cowan et al. (1998), the authors proposed a compiler extension technique called Stack Guard. In each function prologue, Stack Guard inserts a canary value directly next to the return address on the stack. By checking if the value of inserted canary is changed, the buffer overflow attacks will be detected and enter a fail-safe state. However, it is possible to overwrite the return address with keeping the value of canary unchanged.

Another proposed software solution is permitting buffer overflow but to inhibit changing control flow of victim program. PC encoding mechanism (Tyagi and Lee, 2000; Pyo and Lee, 2002) uses semantic encryption to encrypt and decrypt the return address in each function prologue and epilogue respectively, and hence the attacker will not be able to gain control of an attacked system without knowing the encryption key. But, as is well known, Computer hackers are specialists with decryption and knowing the key is not very difficult, which make this strategy not fully secure.

Also another work done by Johns and Kelly (1997) proposes to perform boundary checking on array and pointers in C program. This method can stop the buffer overflow attacks by preventing the injection of a payload and the modification of return address. Although building boundary checking mechanism into the compiler may be considered the ideal solution to the problem of buffer overflow vulnerability, this solution is not effective practically due to the high performance penalty.

Another solution with the same idea for preventing the injection of payload and the change of the return address are shown in Libsafe (Baratloo et al., 1999) as compiler
A software approach for stack memory protection based on duplication

level solution and Libverify (Baratloo et al., 2000) as binary level solution. Libsafe (Baratloo et al., 1999) is proposed to intercept vulnerable library calls at run-time and check for arguments bounds, for example, for strcpy() it checks the length of the source string and also check the upper limit on the length of destination string based on the pointer value of current frame. Libverify (Baratloo et al., 2000) checks the integrity of return address by implements dynamic binary translation. Nevertheless, high overhead is incurred due to adding checking code and applying code instrumentation at run-time.

In addition, there is a different approach which makes the stack area a non-executable space. For example, the Solar Designer allows the buffer overflow on the user stack but prevents executing attack code since the user stack is non-executable. Mainly, control attacks are not addressed because they do not depend on overwriting adjacent local variables but on changing the control flow.

2.2 Hardware solutions

Although hardware solutions may show more resistance against buffer overflow attacks in some cases, it is not a favorable by computer users. All current hardware solutions require a specific hardware or configuration or the both to be implemented.

One of hardware protection techniques against control flow attack is proposed in Francillon et al. (2009). The authors introduced a split stack technique. They separate the stack into data stack and return stack to save the return addresses and protect them in hardware against attack modification. Also, to provide more protection for return stack, they proposed an instruction-based memory access control to prevent the attacker who able to write on a specific memory location. These techniques take care of control attacks but the non-control attacks that depend on buffer overflow to overwrite adjacent variables are not addressed. Moreover, the hardware and architecture modifications needed to implement these techniques are very expensive.

Another hardware solution to protect return address on stack is SmashGuard. It relies on changing the semantic of call and ret instructions in the instructions set architecture. The new call instruction store a copy of return address in memory segment while the new ret instruction check the return address on stack with the stored return address before return. In the case where two values do not match, the processor raises a hardware exception and the execution is terminated.

Similarly to SmashStack, a hardware mechanism proposed in Xu et al. (2002). The authors modify the call and ret instructions to save and read control flow information from the proposed return stack. Although, this return stack is located far away from normal stack, it is still possible to overwrite the return address by double corruption attacks (Alexander, 2005). CASH is another scheme that exploits the hardware segmentation of Intel to perform array bounds check to provide protection against buffer overflow. Another possible solution performs stack checking based on register windowing approach of the SPARC processor is StackGhost. It requires restoration the operating system.

3 Background

In this section, we present a theory background for stack basics, buffer overflow problem, stack-based attacks, and buffer overflow defenses.
3.1 Stack basics and layout

The stack segment of process memory is considered because it is an easy window and common in hacking a program. Moreover, until now there is no solution that has been developed yet to protect stack completely from stack-based buffer overflow attacks. Figure 2 shows the content of the stack frame in a stack segment of memory.

Figure 2  (a) Process memory layouts (b) Stack layout (see online version for colours)

![Figure 2](image)

3.2 Buffer overflow problem

Buffer overflow is a type of vulnerability caused by a bug in an application and it is considered as one of the most widespread software vulnerabilities exploited, although many studies and researches were introduced against buffer overflow attacks. Buffer
overflow is a program case where more data is written in an allocated buffer (i.e., array content) than it can handle and may allow an attacker to change the execution flow of the hacked program. It may happen on stack or heap memories. Since most of the buffer overflows vulnerabilities are occurred on stack (Xu et al., 2002), our research is only focused on stack overflow.

One of the most impressive ways to gain the execution flow of the program and even the system is by modifying the return address of a function or a program. Stack overflow problem is caused while as stack grows down, a buffer grows up in the stack area during runtime. As a result of stack overflow, the old base pointer and return address are overwritten and the execution flow of program is converted to new value of a return address. Figure 4 shows an example of stack overflow problem. Now, overflowing the old base pointer to point to a fake stack frame with a return address pointing to an attack code is one form of stack buffer overflow vulnerabilities. But, overflowing the return address to directly point to an attack code is more popular; hence we only studied and solved the problem of exploiting return address on the stack.

Figure 4  Stack overflow example (see online version for colours)

In the code shown in Figure 4, notice that the buffer size in the function’s definition is smaller than the loop counter upper limit; hence a stack buffer overflow is occurred. In addition to this way, the stack buffer overflow attack can be done by exploiting the unsafe functions in the standard C library such as: strcpy(), strcat(), getwd(), gets(), fscanf(), scanf(), and sprintf() (Wilander and Kamkar, 2003). After filling the buffer with ‘A’ values, the code will overwrite the old base pointer and return address and even argument C with ‘A’. As a result, this overflow the new value of old base pointer and return address with 0×41414141 values (the ASCII value of ‘A’ is 41). This cause segmentation fault since 0×414141 is outside of the process memory address space. However, if the attacker injects an arbitrary attack code in the executable stack area instead of ‘A’ and overwrite the return address by address of attack code, an attacker can redirect the execution flow of program to the attack code and may gain the control of the whole system.
Defending against buffer overflow vulnerabilities and attacks can be classified into four basic approaches (Park et al., 2007): writing correct code, non-executable buffer, array bound checking, and code pointer integrity checking.

Writing a correct and safe code requires programmers to use tools that help them identify vulnerabilities in the source code of programs. While these methods and tools are helpful in developing more secure programs, it is impossible to provide a full assurance that all buffer overflows vulnerabilities have been found because of C semantics features (Park et al., 2007). Moreover, such methods are useless when using legacy code and old libraries, which are common practice among software developers. Therefore, for high guarantee against buffer overflow vulnerability, methods that are more protective should be used.

The main idea of non-executable buffers method is to stop executing the code injected by attacker into the hacked program’s input buffers by making the data segment of the hacked program’s address space non-executable. More effectively is to make stack segment only non-executable and preserve most program compatibility.

On the other hand, array bound checking method try to provide protection against buffer overflow attacks. If the array cannot be over flown by excessive data, then array overflow cannot be exploited to overwrite adjacent areas, and thus over tightening the return address. Nevertheless, array bound checking methods check only explicit array references and cannot correctly check array bounds when passing arrays as arguments to functions. Furthermore, array bound checking protects only an available source code of program from buffer overflow attacks.

Code pointer integrity checking methods (Cowan et al., 1998; Baratloo et al., 2000; Cowan et al., http://www.cse.ogi.edu/DISC/projects/immunix; Wagle and Cowan, 2003) check the integrity of a code pointer such as a return address or function pointer prior to using it. Then, if the attacker has the capability to change code pointer, the attack can be detected and the modified code pointer cannot be used. All such solutions require assumptions on the steps to complete attacks and thus are considered not comprehensive enough to target all possible attacks. For example, StackGuard (Cowan et al., 1998) inserts a canary value before and after a return address that is checked for integrity or modification prior to using the return address. This solution fails if the return address is modified using an integer overflow attack rather than a buffer overflow attack.

4 Algorithm and implementation

It is worth to note that a successful attack can lead to a system hijack and so the attacker can compromise the whole system. Unfortunately, all previously proposed solutions as we discussed in section two, that deal with this problem suffer from shortcomings highlighting the need for further research in this vital area.

In this paper, we propose a general software solution for stack-based buffer overflow vulnerabilities and attacks. Our goal is to provide a protection and at the same time maintaining high performance and lowest cost. We create a new patch tool that fixes a wide-range of stack related vulnerabilities in existing applications. Mainly, our solution depends on duplication and randomisation of the new stack locations where the return addresses will be stored and checked.
Our solution is able to detect and prevent any type of attacks against return addresses on the stack. The idea behind our proposed solution is to create random number of copies of the return address on the stack, then allocating random stack location for each copy of the return address, and then start storing these copies from random offset within each allocated stack in each function prologue. Before the function return, all duplicate copies are checked in parallel such that any mismatch between any of these copies would indicate a possible attack attempt and would raise an exception. At run time, the secure program will create random number of instances of return addresses in the stack, called Mirror Return Address Stacks (MRAS), in the heap memory. The numbers of copies are (three to five) and locations of these MRAS structures are randomised based on the process ID, date and time, machine ID. Therefore, the number and locations of these MRAS structures will vary between different instances of the same program.

When a return address is to be pushed into the stack, copies are also pushed into the MRAS structures. When the return address is to be used, all copies are checked for integrity. A discrepancy in one or more of the return address values would indicate a possible attack attempt and would raise an exception. The exception handler may terminate the program, alert the owner, or rollback to a previous state before the attack attempt. In order for an attack to succeed, it must breakthrough three security levels of our approach. In other words, the attacker must be able to identify the number, locations of the MRAS structures, and the offset address within each MRAS structure, and then make changes in all of them. This is an extremely difficult process, and requires the availability of many bugs in the program, which is not the case in most software vulnerabilities reported by CERT. Moreover, since the number of MRAS instances and locations are randomly chosen at run-time, an attack that succeeds on one instance of the application will most likely not succeed on other instances that may have a different number of MRAS copies and locations.

4.1 Algorithm and implementation

Our proposed approach involves creating a new patch tool that fixes stack related vulnerabilities in existing binaries. The input is an assembly file which is extracted from the vulnerable executable binary file using an appropriate disassemble tool. The output is an enhanced assembly file with improved stack protection and security. This assembly file can then be converted back into an executable binary using the assembler. Figure 5 shows the main procedures to implement our proposed solution from a vulnerable binary file into a more secure binary or executable file passing through three phases as shown in the following:

- Phase 1: Disassembling the binary file using the Objdump disassemble tool. The output of this phase is an assembly file.
- Phase 2: Apply our patch to the extracted assembly file. The patch analyses the assembly file and adds the necessary protection code to implement our solution. The output to our solution is a more secure assembly file.
- Phase 3: Re-create the binary file using an assembler such as GCC patch tool.
Objdump is a tool used to display information about object files (Kharbutli and Samma, 2009). It can be used as a disassemble tool as we did in our work to view binary file in assembly form. Objdump mostly used by programmers who are working on compilation tools, in contrast to programmers who just care about compile and run their programs.

4.2 Our patch tool

We create a new patch tool that fixes a wide-range of stack related vulnerabilities in existing applications. The input to our patch tool is an assembly file which is disassembled from the vulnerable binary file using objdump disassembler. The output is an optimised assembly file with improved stack protection and security. This assembly file can then be re-compiled into an executable or a binary file using the assembler.

Our new patch tool is implemented and written to work as follows (Figure 6):

- Step 1: Convert the input assembly file of objdump format to a valid assembly file for compilation.
- Step 2: Apply our three levels of security based on randomisation and duplication to store return addresses.
- Step 3: Insert the protection instructions before each call and return instructions in the assembly file.

Figure 5  Main procedures to implement our proposed solution (see online version for colours)

![Diagram](image-url)
4.3 Convert objdump to x86 assembly file for compilation

As it has been discussed above, the input to our patch tool is the output assembly file from Objdump disassembler. At this point, we faced a difficult problem since the output assembly file of the objdump is not suitable for the purpose of compilation because of added headers, addresses, and labels, so it cannot be used to re-create or convert into a binary file. Therefore, we convert the assembly file of objdump format to an x86 assembly file which is valid for compilation. The conversion has been done on several stages.

Since objdump uses the address of a variable instead of its name as operand in the assembly instructions, and x86 uses the name of the variable, we replaced the address of the variable by its name. The first stage of conversion involves storing the variables of BSS and TEXT segments that exist in a dynamic table. The variables are saved in linked-list called VARIABLE as shown in Figure 7. Each address that is used in an assembly instruction during the code is compared with all addresses that are stored in VARIABLE linked-list and if it exists then the address is replaced by the name of variable of matched address. For the variables of BSS segment, we only insert a definition for each variable at the end of the output file that contains the .comm keyword along with the variable name, size, and alignment.

Figure 7 Declaration of variable structure (see online version for colours)

```c
typedef struct variable
VARIABLE; //structure to save functions
struct variable
{
  char *name;   //variable name
  char *address; //variable address
  char section; //first character of variable section
  char *size;   //size of variable
  int length;   //length of variable name
  int def;      //flag if variable is already defined or not yet
  VARIABLE *next;
};
```

In the second stage, the input file is scanned to look for the section that includes all important labels that are used by the instructions during the code. This section in Objdump file is called .rodata section. The column on left-most contains the start addresses of .rodata section.

In x86 assembly file, each label has a unique ID, therefore we used counter of three digits for labels IDs starting with LC000. To derive the needed labels for x86 assembly file from .rodata section in objdump file, we save the first start address as START_ADDRESS variable and the last address as END_ADDRESS which will be used later. In addition, the hexadecimal values are stored in HEXA_VALUE array and the whole characters in the right-most column are stored in the DATA_SECTION array. In objdump, there are two cases of using addresses that belong to .rodata section:
• Case 1: Addresses that are referenced to labels of type string. These addresses are preceded by $ character.

• Case 2: Addresses that are referenced to labels of type long. These addresses are used without any addition.

The implementation is as follows: the .text segment is scanned using input assembly file for all addresses that belong to .rodata section. For labels of case 1, the .string keyword is added which means it is a string variable and (”) to determine the start of the string. Next, the characters are copied one by one from DATA_SECTION array beginning from character of found address until a NULL character is found, hence, the copying is stopped and the string is ended with (”). Also, index of DATA_SECTION array that store the first character in the label is saved in LABEL_INDEX array. For labels of case 2, there are two parts for each label. The .long keyword is added before every part which means it is a long variable. Then, the 16 hexadecimal values beginning from hexadecimal value of found address are saved from HEXA_VALUE array to temp array. First 8 digits are converted to decimal value and copied to the first part of the label and the other 8 digits are converted also to decimal value and copied to the second part of the label. Also, index of HEXA_VALUE array that store the first hexadecimal value in the label is saved in LABEL_INDEX array. Figure 8 shows the labels in x86 assembly file that derived from .rodata section.

Figure 8  The labels in x86 assembly file (see online version for colours)

```
LC000: .string "the original matrix is :"
LC001: .string "(\%d,\%d)=%f	"
LC002: .string "THE DETERMINANT OF THE ABOVE MATRIX IS: %f 
LC003: .string "the result matrix is:" .long 0
LC004: .string "(\%d,\%d)=%f	"
LC005: .long 0
LC006: .long 0 .long 10745291 .long -6320914
```

Another difference between ×86 format and objdump format is in the way to point to string or long variables since ×86 format uses labels while objdump format uses addresses. Therefore, to complete the conversion, the next stage implies replacing the addresses that belong to .rodata section with labels.

In objdump assembly file, before calling built-in function that used to print a statement or a value of variable on screen, the address of the first character in this statement should be stored in the stack. As demonstrated in Figure 9, the highlighted
address 0x8048563 is the address of the first character of a statement that will be printed on the screen by printf() function. The statements in the data section are terminated with NULL character.

**Figure 9** How the statements are printed in objdump format (see online version for colours)

```
Contents of section .rodata:
08048554 03000000 01000200 00000000 25640066 ..........%d %d
08048564 6163746f 7269616c 20616620 25642069_aecoral of %d
08048574 73202564 0a00
```

As has been discussed, in the second stage when the data section is converted to labels, four variables are saved:

- **START_ADDRESS** is the start address of characters in data section.
- **DATA_SECTION** array of type char contains all characters in data section.
- **HEXA_VALUE** array of type char contains all Hexadecimal values in data section.
- **LABEL_INDEX** array of type int contains the index where the first character in each label is stored in DATA_SECTION array.

For example, when applying the second stage on data section that appears in Figure 9, the three variables will be as following:

The operation of replacing the addresses of data statements with labels during this stage is implemented to work as follows:

- The address of statement is stored in **STATE_ADDRESS** variable.
- The difference between **STATE_ADDRESS** and **START_ADDRESS** is calculated to get **DIFF** variable.
- Search for **DIFF** value in **LABEL_INDEX** array.
- There must be a match since all statements were stored in labels during first stage. Therefore, the replacing label is the index of **DIFF** matching value in **LABEL_INDEX** array.
In Figure 9, the highlighted address will be replaced as follows:

- **STATE_ADDRESS**: 8048563
- **DIFF** : 8048563 – 8048554 = 0xff = 15

The index of 15 is 1 in **LABEL_INDEX**, hence the replacing label is ((LC001)) and the code in Figure 8 is converted in x86 to:

```
main:

L8048469: movl $LC001,(%esp)
L8048470: call printf@plt

L8048482: ret
```

Next stage involves copying main() function and all other user functions codes from the input file. Figure 10 declares the linked-list **FUNCTION** that employed to store all information related to the function that is required for this stage implementation.
A software approach for stack memory protection based on duplication

In contrast, in ×86 it is not acceptable to have the address before the instruction and it gives an error during compilation stage. Since these addresses are used by jump instructions, they cannot be removed. On the other side, deleting them will cause a problem in ×86. To solve this problem efficiently, our patch tool simply adds a letter ‘L’ before each address preceded instruction and each address used in jump instructions. In this case, the addresses are converted to labels and also the jump address is converted to a jump label which is applicable in ×86 format.

4.4 Applying the three security levels based on randomisation and duplication

In the previous phase, our patch tool solves the problem of compilation by converting the input file from objdump format that cannot be compiled into an ×86 compatible assemblers. Nevertheless, our generated assembly file is still vulnerable to stack-based attacks and no protection is provided yet to the input application.

As we discussed earlier in this paper, stack-based attacks exploit vulnerabilities in software programs such as buffer overflow and especially programs written in C/C++ language since it does not perform boundary checking of arrays. Such attacks aim at overwriting the return address by an address that points to a malicious code, which may cause serious problems. For example, the attacker can gain a control of a whole system. Therefore, to protect the vulnerable programs against stack-based attacks and prevent executing the malicious codes even if they are already inserted into memory or stack, we propose an efficient solution that is implemented and discussed as follows:

Our patch tool provides the generated ×86 assembly file from previous phase with three levels of security to protect return addresses comprehensively in the stack against all types of return address attacks. These security levels are based on randomising and duplicating the locations where the return addresses will be stored. Figure 11 demonstrates the code of our patch tool that implements those security levels. This function is inserted at the beginning of the generated ×86 assembly file after it is converted to x86 assembly instructions and it is called in beginning of main () function.
At run time, a random number between (1 and 5) is saved in \textit{nCopies} variable. Then, \textit{nCopies} of instances of return address stacks are created, called Mirror Return Address Stacks (MRAS), in the heap memory using \texttt{sbrk()} system call. Every MRAS has size of 10k bytes. Finally, the offset of each MRAS is also randomised such that the start location to store return addresses is different among all mirror stacks. The number \textit{nCopies}, locations, and offsets of these MRAS structures are randomised based on the process ID, date and time, machine ID etc. Therefore, the number, locations, and offsets of these MRAS structures will change between different instances of the same program.

In order for an attack to succeed, the attacker must be able to identify the number, locations and offsets of the MRAS structures, and then modify all of them with the same required address that is used to overwrite the return address on stack. This is an extremely difficult process, since the number of MRAS instances and locations are randomly chosen at run-time, an attack that succeeds on one instance of the application will most likely not be succeed on other instances that may have a different number of MRAS copies and locations. This prevents the spread of an attack.

4.5 Return address protection

The idea behind creating a multiple instances of MRAS structures is to use them to store a copy of the return address in each MRAS every time the function is to be called, and also use them to check the integrity of return address by comparing all copies of MRAS structures with the current return address value in the stack when the function returns. Figures 12 and 13 present our patch tool to implement both functions of storing return address in function prologues and checking integrity of return address in function epilogues.

The functions \texttt{storeRA()} is shown in Figure 12 are converted to \times86 assembly instructions and inserted in the beginning of the generated \times86 assembly file. When a return address is to be pushed onto the stack, copies are also pushed onto the MRAS structures. In the function prologues, our patch tool applies the following steps:

- Get the value of the return address that will be pushed into the stack by inserting assembly instructions in the beginning of \texttt{storeRA} function as demonstrated in Figure 13.
- Search for ‘call’ instructions in \times86 assembly file.
For each found instruction, it is checked if the called function is either built-in or user function.

If the called function is a user function, ‘call storeRA’ instruction is inserted immediately before the matching call instruction as shown in Figure 13.

Figure 12  Functions to store and check return address (see online version for colours)

```c
//store return address in all MRAS in function prologues
storeRA()
{
    for(z=0;z<nCopies;z++) //for each MRAS structure
    {
        *(stack_pt[z])=RA; //store return address in current pointed location
        stack_pt[z]++;  //increment the pointer to next location
    }
    return;
}
//--------------------------------------
//compare all stored copies of last return address with current one on stack
restoreRA()
{
    for(z=0;z<nCopies;z++) // for each MRAS structure
    {
        stack_pt[z]--; //decrement pointer to previous location
        if(RA!=*(stack_pt[z])) //compare return address on stack with stored copy
            break;  //if no match break the loop
    }
    if(z!=nCopies) //not all copies are equal: attack attempt
        {printf("attack attempt\n");
            return;  //terminate program
        }
    return;  //return successfully
}
```
Figure 13  Protection instructions to store return address in function prologues (see online version for colours)

```plaintext
storeRA:
  pop  RA  //pop top of stack to store current value of program counter in RA
  push RA  //push it again to stack so keeping the program state unchanged
  add $5, RA  //add 5 to RA, since current program counter is the address of call instruction and the return address will be the address of next instruction which is larger by 5 bytes (the size of call instruction)

main:
L804844d:  mov    -0xc(%ebp),%eax
L8048450:  mov    %eax,(%esp)
call  storeRA // store return address before each call
L8048453:  call   factorial
L8048458:  mov    %eax,-0x8(%ebp)
```

Figure 14  Protection instructions in function epilogues (see online version for colours)

```plaintext
L8048427:  leave
  pop  RA  //pop top of stack to store current return address in RA
call  restoreRA
  //checking the integrity
  push RA  //push it again to keep program state unchanged
L8048428:  ret
```

In the function epilogues, when the return address is to be used, all copies of MRAS structures are checked for integrity. A discrepancy in one or more of the values would indicate a possible attack attempt and would raise an exception. The exception handler may terminate the program, alert the owner, or rollback to a previous state before the attack attempt. The implementation of our patch tool in function epilogues is as follows:
Search for ‘ret’ instruction in main ( ) function and all other user functions.

Insert the protection assembly instruction exactly before each matching instruction as presented in Figure 14. These instructions are responsible for getting the current return address on stack and check its integrity with stored copies of return address.

Here, our patch tool finishes the work and outputs a secure ×86 assembly file that is can be compiled without affecting the functionality of input application. The next and final phase is the re-compilation of output assembly file using GCC patch tool as an assembler to get back the same input binary file provided by our efficient and protection mechanism against return address attacks.

5 Evaluation

As a proof of concept, our software tool is tested using several known programs to verify the performance overhead. In next section we used the most common stack-based attacks to verify the correctness and the level of protection of our software tool.

5.1 Performance

To test the performance overhead we ran several micro benchmarks. We collected the results of running programs instrumented with and without the code that implements the stack protection MRAS. All tests were run on a single machine [Intel(R) Core(TM)2 Duo CPU T5750 at 2.00 GHz, 3064 MB RAM, GNU/Linux fedora9]. The GCC compiler has been used to instrument assembler code with new instructions. The benchmarks show that this implementation experiences the significant factor of between 1% to 170% slow-downs as declared in Table 1. The result of run time in Table 1 is the average of 50 runs for each program. There was no need to do more run because they were close in 1% to each other. The memory overhead of our tool is in range of 30 to 50 KB such we create 3 to 5 new stacks at size of 10 KB to protect return address.

<table>
<thead>
<tr>
<th>Micro benchmark</th>
<th>Runtime without MRAS(s)</th>
<th>Runtime without MRAS(s)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble sort for 100,000 numbers</td>
<td>72.063</td>
<td>73.462</td>
<td>1%</td>
</tr>
<tr>
<td>Sorting algorithm for 20,000 number by 6 algorithms</td>
<td>53.256</td>
<td>54.073</td>
<td>1%</td>
</tr>
<tr>
<td>Factorial for 10,000,000 numbers</td>
<td>109.851</td>
<td>109.874</td>
<td>16%</td>
</tr>
<tr>
<td>Matrix inverse of 11 × 11</td>
<td>49.127</td>
<td>74.427</td>
<td>31%</td>
</tr>
<tr>
<td>Fibonacci for 100,000 numbers</td>
<td>54.661</td>
<td>146.623</td>
<td>170%</td>
</tr>
</tbody>
</table>

The MRAS implementation is affected by a different increase of the execution time when compared to reference time in different programs as shown in Figure 15. That is mostly due to the number of function calls in the program such that the overhead is increasing as the number of function calls is increasing in program. In addition, the programs that contain recursive function have overhead higher than programs that do not contain recursive function. We conclude from this, that this type of protection instructions, are technically feasible and fast.
Table 2 shows more details and declares the reasons of the wide variety of overhead between programs. As shown in the table, Bubble sort and Sorting algorithms programs have a very small overhead approximately zero while Matrix Inverse and factorial programs have a little higher overhead but Fibonacci program have a very high overhead. The reasons are limited in number of called functions in each program and the existence of recursive function in the program and its number.

As we discussed in our implementation in section four, we add our protection instruction before each called function and in the end of each called function. Therefore, as the number of called function increases in the program, the accumulative execution time of program will increase too.

The high overhead of Fibonacci program is due to the high number of recursive called functions that are needed to calculate the Fibonacci number. On other side, although the high complexity of sorting algorithms program, it has 1% overhead only. The absence of recursive function is the reason and this is the proof that our protection instructions have small overhead.
5.2 Protection against stack-based attacks

To verify the protection level of our software tool, we tested it using the major known types of stack-based attacks against return address as follows.

5.2.1 Integer overflow attack

Integer overflow attack replaces return address on stack by a desired address that points to malicious code in memory. Figure 16 shows an example code for integer overflow attack. When the attacker inserts the location of return address on stack as ‘id’ and the desired address as ‘score’, the return address will be overwritten by desired address. Then, when the function uses return address to return, the program will continue at malicious and that may cause critical problems.

Figure 16 Integer overflow attack (see online version for colours)

```
#include<stdio.h>

void rootFunction()
{
    printf("I am root\n");
}

void userFunction()
{
    int A[10], id, score;
    printf("I am user\n");
    printf("Enter your ID and score\n");
    scanf("%d %d", &id, &score);
    A[id] = score;
    Return;
}

main()
{
    userFunction();
    printf("Returned to main\n");
}
```

In our software tool, that is impossible to happen. The return address could be overwritten but the program will never continue at malicious code because at the end of each function and before using return address, our protection instructions compare current return address on stack with the one that saved before calling the user function and any mismatch between current return address and any copy of the saved return address will indicate an attack attempt and terminate the program immediately.

5.2.2 Buffer overflow attack

The buffer overflow attack overwrites a buffer on the stack to replace the return address. When the function returns, instead of jumping to the return address, control will jump to the address that was placed on the stack by the attacker. The code in Figure 17 declares an example of buffer overflow attack. The size of ‘Name’ is eight bytes, but the code
write ‘A’ 40 times which will overwrite the bytes below Name buffer. Return address will replaced too by 0×41414141.

Figure 17  An example of buffer overflow attack (see online version for colours)

```c
#include<stdio.h>
void rootFunction()
{
    printf("I am root\n");
}
void userFunction()
{
    char Name[8];
    for(i=0; i< 40; i++)
        Name[i] = 'A';
    printf("I am user %s\n", Name);
    return;
}
main()
{
    userFunction();
    printf("Returned to main\n");
}
```

Without our MRAS protection the program will continue at that address, but in our tool and with MRAS protection the program will indicate an attack attempt and terminate the program immediately because the return address that was saved before user function is called will be different than current return address on stack which is 0×41414141 in this case.

6 Conclusions

In this paper, we proposed, implemented, and tested a general approach to achieve full protection for stack return addresses of all types of attacks. The main idea in our solution is to create a random number of copies of stack return addresses and store them at random locations. When a pointer is used, all copies are read and compared. In order for an attack to succeed, the attacker must know the number of copies, their locations, and be able to modify all of them simultaneously. This is an extremely difficult process. The proposed software tool was tested using several kinds of known micro benchmarks to verify its performance and the results show a small overhead comparing to reference time. Moreover, the software tool proved a high protection against integer overflow attack and buffer overflow attack.
A software approach for stack memory protection based on duplication

References


