ENGINEERING JOURNAL

Survey of Protections from Buffer-Overflow Attacks

Krerk Piromsopa[1,*] and Richard J. Enbody[2]

1 Department of Industrial Engineering, Faculty of Engineering, Chulalongkorn University
Bangkok 10330 Thailand
2 Department of Computer Science and Engineering, Michigan State University
East Lansing, MI 48824-1226 USA
E-mail: krerk@cp.eng.chula.ac.th[1,*] and enbody@cse.msu.edu[2]

Abstract. Buffer-overflow attacks began two decades ago and persist today. Over that time, a number of researchers have proposed many solutions. Their targets were either to prevent or to protect against buffer-overflow attacks. As defenses improved, attacks adapted and became more complicated. Given the maturity of field and the fact that some solutions now exist that can prevent most buffer-overflow attacks, it is time to examine these schemes and their critical issues. In this survey, approaches were categorized into three board categories to provide a basis for understanding buffer-overflow protection schemes.

Keywords: Buffer overflow, buffer-overflow attacks, function-pointer attacks, intrusion detection, intrusion prevention.

ENGINEERING JOURNAL Volume 15 Issue 2
Received 1 June 2010
Accepted 5 February 2011
Published 1 April 2011
Online at http://www.ej.eng.chula.ac.th/eng/
DOI:10.4186/ej.2011.15.2.31

DOI:10.4186/ej.2011.15.2.31

1. Introduction

Though dated back to the 1988 infamous MORRIS worm [1], buffer-overflow attacks are still widely found. In 2007, buffer overflows made up the majority of the vulnerabilities listed by CERT [2] which allowed execution of arbitrary code, i.e. the most serious vulnerabilities. Though skilled programmers should write code that does not allow buffer overflows, current practice does not guarantee programs that are free from vulnerabilities. This way, no software is completely safe from buffer-overflow attacks. Given the persistence of the attacks, eliminating is considered to be difficulty if not impossible. As operating systems managed to get rid of buffer overflow, applications still suffered from being exploited. Running an application with root or administrator privileges also give devastated impact.

Rather than relying on individual skill, a variety of methods to protect systems from buffer-overflow attacks have been proposed. Nonetheless, a protection is not completely provided by most of them. As protection has improved, attacks have adapted, so an “arms race” developed. For example, the original stack-smashing attack can easily be detected by early protection schemes, but more recent attacks that focus on the heap instead of the stack are being ignored.

Our aim is to develop a basic understanding of buffer-overflow and buffer-overflow attacks, to outline and categorize techniques that have been developed, and to analyze critical issues related to each method. We begin with an overview of buffer-overflow attacks to provide the reader with a rudimentary understanding of the attacks and basic protection mechanisms. Next, we provide a more in-depth look at buffer overflows to provide a foundation for the analysis. Finally, we classify the protection schemes and identify critical issues.

2. Overview

2.1. Basic Example

A buffer overflow (a.k.a. buffer overrun) happens when data written to a buffer overflows into memory adjacent to the allocated buffer. Taking advantage of the corruption of the adjacent memory is the basis of an attack.

To illustrate a basic overflow, consider the following example of a 6-byte long string buffer **X** followed by a 4-byte long integer **Y**. Initially, the string **A** contains only ‘-’ characters and the integer **B** contains the number 9.

X	X	X	X	X	X	Y	Y	Y	Y
-	-	-	-	-	-	0	0	0	9

If a program stuffs too big a word into buffer **X**, the result is an overflow into **Y**. For example, store the string “attackers” into **X**. (Assume that the string is terminated with a null byte: ‘\0’.)

X	X	X	X	X	X	X	X	Y	Y
a	t	t	a	c	k	e	r	s	\0

Although the programmer only modified **X**, the result was that **Y** was modified, too. The modified bits of **Y** will represent some integer other than 5. If **B** is a return address, a function will return to a different place than originally intended.

If some mechanism exists that prevents the overflowing of **X** into **Y**, an attack will fail. For example, if a bounds checking mechanism exists that identifies that the string “attackers” is too big to fit in **X** and stops the operation, a buffer-overflow will be avoided and this attack will fail.

The goal of an attacker is to tailor the modification of **Y** so that it can take the program someplace to compromise system security (assuming that **Y** is an address such as a return address). The defender can either stop the overflow of **X** in the first place or detect the maliciously modified of **Y** later.

The crucial concept of buffer-overflow attack is to modify a control address such as our return address **Y**. By modifying a control address, an attacker can cause the program to go where the attacker wants rather than where the programmer intended it to go.

Here is a piece of C-code that takes input from the command line through the variable "argv" and writes it into a buffer using the *strcpy* function (*str*ing *copy*) without any bounds checking. If crafted carefully, the input string can overwrite a return address by overflowing the buffer on the system stack and modify the location to which the program returns. If a program is executed as a privileged user and the modified address is a shell address, the attacker now has opened a shell in privileged mode and "owns" the computer. (Note: in practice this piece of code cannot satisfy all those conditions so it is not a complete and fully effective piece of attack code.)

```
/* Example of buffer overflow. */

#include <string.h>
#include <stdio.h>


int main(int argc, char *argv[])
{
  char myBuff[10];
  strcpy(myBuff, argv[1]);
  return 0;
}
```

2.2. Sample of Prevention Techniques

Here we provide a few samples of existing prevention techniques so the reader can get a sense of the field, especially with respect to techniques they may have heard about. After this brief introduction we will present the full variety of techniques in full detail.

In the basic example above, we made three points:

- Bounds checking of buffer **A** can prevent a buffer overflow.
- Recognizing that address **B** was modified and taking corrective action can prevent a buffer-overflow attack from succeeding.
- The key concept is preventing the attacker from modifying the program's control data.

The best solution is to write code without vulnerabilities. The whole field of software engineering is premised on writing correct code. In particular, provably correct code should eliminate buffer overflows (as well as most other errors). However, currently these techniques are not widely applied, so vulnerable code persists. In addition, the conditions under which code is run may change so that assumptions used for correctness can become invalid over time.

Some languages such as Java bounds-check their buffers so they can prevent the overflow of buffers. However, Java is not a panacea since the underlying run time system has been attacked with buffer overflows [3, 4]. Other languages offer the option of bounds checking, such as the C++ STL, but many programmers avoid that support because bounds checking everything can be inefficient and selective bounds checking is not used or not used correctly.

It is possible to statically check a program to prevent the use of functions (such as *strcpy* in our example) that do not perform bounds checking. However, substitutes (such as *strncpy*) are not always used correctly, resulting in programs without *strcpy* but with buffer overflows. Compilers can enforce restrictions on specific function usage.

Hardware modifications such as Intel's NX are now built into consumer processors. The concept behind that hardware is to stop maliciously injected code from being executed on the system stack by marking the stack as non-executable (i.e. Non-eXecutable). The stack has been a popular and easy place to put malicious code so the attacker will modify a return address (such as **B** in our example) to be the address of the malicious code on the stack. If the stack is non-executable, such an attack will fail because the malicious code will not be able to execute. This approach "raises the bar" for attackers rather than preventing attacks because attackers respond by using code somewhere other than the stack.

The latest approaches have focused on recognizing that control data (addresses such as **B** in our example) have been modified and prevent jumps to modified addresses. For example, a "canary" word can be placed next to a control word so that overflowing a buffer trashes the canary word as well as the

DOI:10.4186/ej.2011.15.2.31

control word (the analogy is to a canary in a mine). Compilers such as Microsoft's Visual Studio can add such protection. Attackers have circumvented canary words, so this technique qualifies as "raising the bar."

Most recently, hardware modifications have been proposed that tag external data as "tainted" as it enters a process. If that "tainted" data somehow (say through a buffer overflow) is written into a control address, it is easy to recognize (because it is tagged) and an exception can be raised if the "tainted" address is being used as a program counter. Hardware modifications have the advantage of being harder, possibly impossible, to circumvent.

3. Fundamentals

We begin by establishing the fundamentals of buffer-overflow vulnerabilities, discussing the associated attacks, and describing variations. Later, we provide a more precise definition of buffer overflows and their attacks and show the critical role of control data in attacks. We then show how the concept of preserving the integrity of control data, particularly with respect to external input, leads to the latest, and most effective, buffer-overflow prevention techniques.

There exist attacks on variables (non-control data) achieved through overflowing of buffers, but these are not what most people think of as "buffer-overflow" attacks, since in most cases corrupting control data results in a more virulent attack. A related attack overflows data values, usually integers. Most attacks that overflow data values are a prelude to another attack—usually buffer overflow—because the data overflow attacks guard data. Nonetheless, they are not covered here.

3.1. Necessary Conditions

Several researchers (e.g. [5, 6, 7, 8, 9, 10, etc.]) have posited that there are two necessary conditions for buffer-overflow attacks to be successful: (1) injecting malicious code and (2) redirecting the program control flow to execute that code. However, injecting malicious code is not necessary since the code utilized in the attack can be resident code found in shared libraries. For example, jumping to resident shell code while in privileged mode is sufficient for a successful attack. Therefore, we claim that only the second condition is truly necessary: redirection of control.

Techniques that only protect systems from the injection of malicious code will fail as general protection mechanisms. We will point out such schemes, but will focus on the issue of redirection of control.

3.2. Buffer Overflow

We begin with a definition of buffer overflow (from the Webopedia Computer Dictionary [11]).

Definition 1: Buffer Overflow

> The condition wherein the data transferred to a buffer exceeds the storage capacity of the buffer and some of the data "overflows" into another buffer, one that the data was not intended to go into.
>
> Since buffers can only hold a specific amount of data, when that capacity has been reached the data has to flow somewhere else, typically into another buffer, which can corrupt data that is already contained in that buffer. [11]

When necessary conditions are met, buffer-overflow attacks may result in a serious system security breach. The seriousness of attacks varies. It can be as simple as modifying a variable, accessing unauthorized memory (segmentation fault), or executing attackers' code. Based on the definition of buffer overflow, we define the buffer-overflow attacks in Definition 2.

DOI:10.4186/ej.2011.15.2.31

Definition 2:

A **buffer-overflow attack on control data** is an attack that (possibly implicitly) uses memory-manipulating operations to overflow a buffer, which results in the modification of an address to point to malicious or unexpected code. [12]

Though a buffer-overflow may occur with any location in memory, we use the term "buffer-overflow attacks" to specifically refer to attacks on control data.

Observation: Data passed from another domain (machine, process) is always used to overflow a buffer—hence its malicious nature.

Initially, return address was a target, but later the targets also include other control data (e.g. function pointers, jump table). Nevertheless, the eventual result will change the program to execute the injected malicious code. When buffer overflow is not being used as an instrument for modifying the address, other type of attack such as a race condition, a Trojan horse may give a similar result.

With this observation, we summarize a ***buffer-overflow attack*** as

an attack caused by overflowing a buffer with data from another domain that results in a malicious or unexpected behavior of a program.

This notion is not originally new. For example, Howard and LeBlanc propose in their book that "**All input is evil until proven otherwise**" [13]. With this concept in mind, an ability to detect and validate input, especially input that is eventually used for control, would be an insightful solution against buffer-overflow attacks. More concepts about prevention will be discussed later in section 4.

Using the concept that inputs are the origin of attacks, a key component for detecting and validating the input is metadata. Metadata is additional information on the properties of data. It can be one or a combination of: type descriptor, guard value, encoding key, redundancy copy of data, tagged value, or even programming logic. This concept is also not new. Metadata has been discussed as an instrument for detecting and preventing buffer-overflow attacks [14]. We use the variety of metadata and its management as a way of classifying buffer-overflow prevention schemes.

Note that metadata can also be used for other purposes, e.g. tagged architectures such as Symbolics [15] or tagged operating systems such as [16]. However, we only focus on the use of metadata for buffer-overflow attack prevention.

In summary, a malformed input is being used to overflow a buffer causing a malicious or unexpected result, buffer-overflow attacks. To prevent the attack, some metadata is necessary. Before getting to the details of schemes for handling buffer-overflow attacks, we will delve further into details of attacks and their variations.

3.3. Sample Attacks and Variations

Control data and local variables are two main targets of buffer-overflow attacks. Control data is the target of majority of attacks, so is the focus of prevention schemes. There are several types of control data: function pointers, return addresses, and jump slots. Since the locations of return addresses are deterministic, they have been the primary target. Other control data is targeted by more advanced attacks. Attacks on return addresses are usually referred as first-generation attacks and those on others as second-generation attacks [17].

We begin with an algorithm for performing an attack:

```
// Attack algorithm
1. Find a vulnerable address.
2. Find a buffer near a vulnerable address.
3. Pass an argument to the buffer to cause it to overflow over the
   vulnerable address.
4. Within the buffer craft the data so a new address overwrites the
   vulnerable address.
5. Wait for a (privileged) function to jump to the new address.
```

DOI:10.4186/ej.2011.15.2.31

Elias Levy (a.k.a. Aleph One) [18] provided the first step-by-step description of how to construct a buffer-overflow attack in 1996. In his paper, he coined the term “Stack Smashing” to refer to plastering malicious code and its address to set up an eventual overflow of a return address on the stack.

Figure 1 (taken from Howard [19]) is an example of stack-smashing. In this example, attackers will pass a malicious code and copies of buffer’s address as a parameter to the vulnerable program (through argument *p*). A strcpy function (buffer manipulation function) will overflow the return address of the function with the address of the buffer (containing shell code). The return instruction will use the address of the target buffer as a return address and eventually jump to execute that code. Function pointer *fp*, an additional attack vector, are presented but not used. The modification of the return address (replaced by the address of the target buffer) is the critical component of this attack.

To illustrate an advanced buffer-overflow attack, a variation of Hannibal Exploit [20], **an example of multistage buffer-overflow attack** that can bypass most software buffer-overflow solutions, is presented in Fig. 2. The fundamental of a multistage buffer-overflow attack is a vulnerable pointer that can be modified to refer to arbitrary memory (i.e. a buffer overflow near a nifty pointer). In the first step, the attackers modify a pointer (by overflowing) to point to a specific location (e.g. a function pointer or a jump slot). The second step is to store designated address at the target. That is, attackers can create arbitrary pointer (first stage) and modify a memory location with any address (second stage). Once modified, a latter jump using that target will run code specified by attackers. In particular, the program will be redirected to the attacker’s malicious code. For example, modifying the jump slot of a privileged program to target to shell code allows attackers to spawn a privileged root shell is an example of this scenario of attack.

It is worth clarifying the usage of a jump table. Given that the *printf* binary is a part of standard shared library, a slot in the jump table is used as a pointer to the entry of *printf* in memory. A call to this function in users’ program indexes the table for *printf* slot and then jumps to specify address.

Attacking this vulnerable program requires two stages of buffer overflow. First is overflowing the *ptr* pointer to point to the *printf* slot (1) by specifying the address of this slot in jump table in *argv[1]* . The strcpy routine, by copying *argv[1]* into *buffer*, will replace *ptr* with the *printf* address slot. In the figure, the pointer *ptr* will be changed from ‘buffer’ (arc labeled ‘Before’) to the jump slot (arc labeled ‘After). Once the slot is reachable, attackers can simply alter the value.

Supposedly an address of resident shell code, referred as *residentcode*, has been determined. Once attackers managed to overwrite a slot in the jump table with the address of *residentcode*, a subsequent call to *printf* would change to execute *residentcode,* the shell. Assuming that it is a privileged program with privileged mode; the attacker would have access to a function root shell. (More details can be found in [21]). This attack gets around most buffer-overflow protection schemes. Less obvious is modifying handling vector of some solutions to bypass the handling routine. This allows us to circumvent some software solutions.

Additional attack that cannot be detected by most schemes is a pointer attack. For example we can copy from one memory location to another by modifying source and destination pointers, Fig. 3 shows an attack on both pointers. Attackers can use the first ***strcpy*** to overflow the ***src*** and ***des*** pointers with arbitrary address. In this case, ***src*** is replaced with the address of valid control data (e.g. function pointer of a shell code); and ***des*** is replaced with the address of target pointer (e.g. a slot in the jump table). With the next ***strcpy*** in place, attackers can replace the jump slot with a known local entry without being detected. A possible solution to this type of attack is to encode pointers.

There exist printf vulnerabilities [12], malformed formatting instructions that result in modifying arbitrary memory similar to buffer overflow. However, such attack is not a buffer-overflow attack. Some buffer-overflow protection schemes are useful for this class of attack as well. However, such attack may occur to any variable. To our knowledge, exist protection schemes does not cover arbitrary data. Fortunately, a lexical analysis is sufficient for detecting this type of vulnerabilities. (Note that lexical analysis is not sufficient for buffer overflows.)

DOI:10.4186/ej.2011.15.2.31

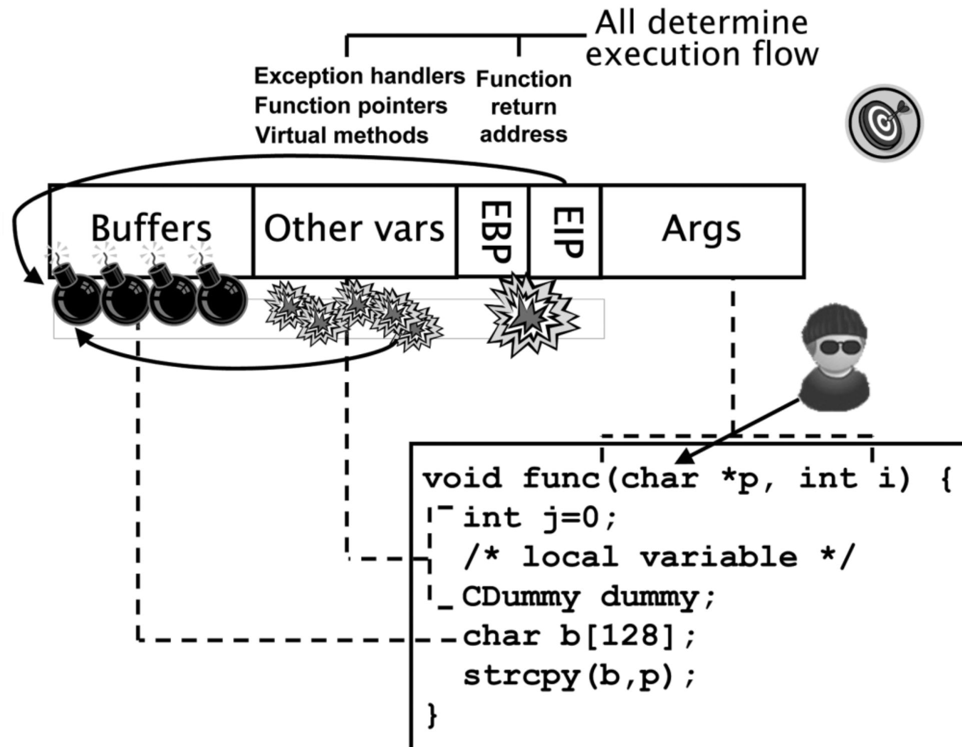

Fig. 1. Stack Smashing (Taken from Howard [19]).

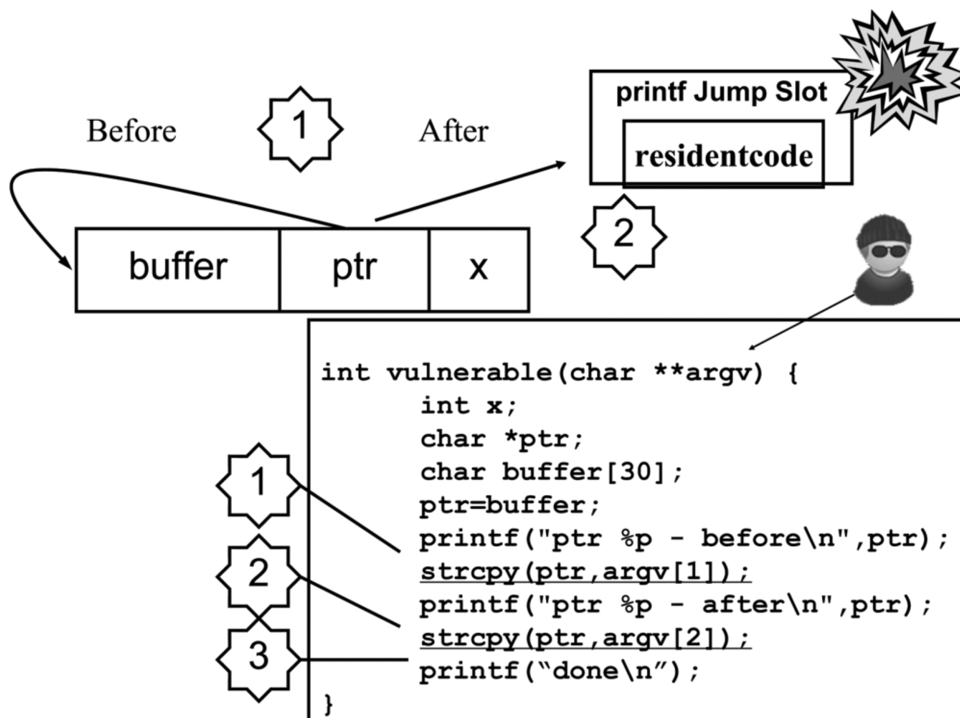

Fig. 2. An example of a multistage buffer-overflow attacks.

DOI:10.4186/ej.2011.15.2.31

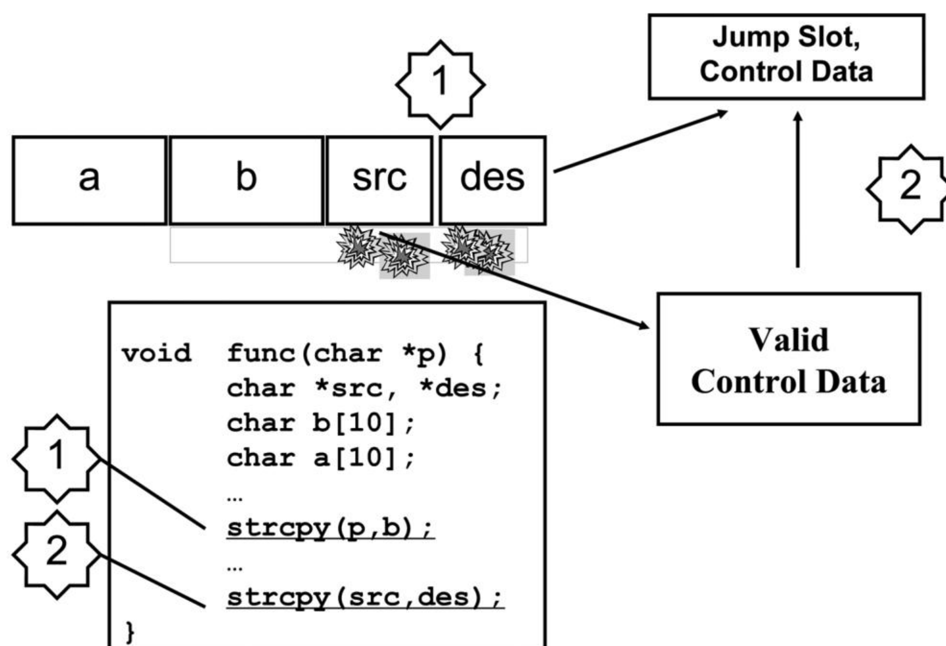

Fig. 3. Buffer-Overflow Attacks on Pointers.

"Integer overflows", also referred as "integer arithmetic" attacks, are presented as a variation of buffer-overflow attacks [22]. The main idea is to attack the arithmetic used for validating the size of buffers (bound checking). With invalid calculation, accessing the wrong-sized buffer is likely to result in buffer overflow. This type of attack can be protected by more robust schemes.

The classic password attack [23] serves as an example of attack on local variables. By overflowing a variable containing stored password, an arbitrary password can result in privilege access. There is no control data involved in this attack. Hence, it cannot be prevented with existing protection schemes.

4. Prevention

Up to this point, the malicious behaviors of buffer-overflow attacks have been unveiled. The necessary conditions for preventing buffer-overflow attacks will further be discussed in this section.

From our basic buffer-overflow example, we saw that preventing the actual overflow (e.g. through bounds checking) is an obvious prevention scheme. Though preventing the overflow sounds simple, in practice, it has turned out to be difficult to implement—hence the continual appearance of buffer-overflow attacks. The problem persists because there is no general mechanism in use across languages in use that can perfectly specify such capacity. The persistence of overflows motivated a different approach.

Because preventing the overflow of a buffer is not working in practice, researchers decided to try focusing on the address (**B** in our basic example). If malicious manipulation of the address can be identified, we can prevent the jump to that address. That is, we can stop the attack right at the moment that the attacker actually takes control.

The critical observation is that the address that the attacker inserts comes from outside the current running process. If we can identify external data, we can recognize that a control address now contains external data. External data in a control address is bad because someone else can now control your program flow.

It would be nice to preserve the integrity of control data by not allowing external data (i.e. from outside this process) into it. However, it is equally effective if we *recognize* that external data is in the control word and prevent a jump to it. From our basic example, we would recognize that **B** now contains external data, so do not jump to that address.

To summarize: a buffer-overflow attack redirects the control flow of a program to execute malicious code. An address must necessarily be modified to achieve this objective. Should attackers fail

to modify the address, the attack fails. Should a system detect the modification of the address, we can recognize and stop buffer-overflow attacks.

5. Protection schemes

Up to this point, we have established a basis for understanding buffer-overflow attacks, so we can now examine current approaches to protect against them. We partition the approaches into four broad categories: correct code, static analysis, dynamic solutions, and isolation. Approaches that allow programmers to vulnerable code related to buffer-overflow attacks are static analysis. Approaches that guard or monitor target data or source of buffer-overflow attacks are dynamic solutions. Approaches that do not stop the attacks but seek to limit the damage caused by the attacks are isolation.

5.1. Correct Code

The whole field of software engineering is premised on writing correct code. In particular, provably correct code should be free of known vulnerabilities. For example, Microsoft has recently devoted significant resources toward creating code without exploitable vulnerabilities—a subset of totally correct code. Their effort, called the Security Development Lifecycle [13], represents how one of the largest software companies implements the development of secure code. Since these techniques are not widely applied, vulnerable code persists.

Also, correct code is not a panacea. For example, the conditions under which code is run may change so that assumptions used for correctness can become invalid over time. A recent example is the Windows WMF vulnerability (CERT VU#181038 [2]), which allows an attacker to work through image metadata to execute arbitrary code, i.e. take complete control. The path is through a Graphics Device Interface (GDI) function (SetAbortProc), which first appeared in Windows 3.0 in 1990 and provided important functionality in that context. Later versions of Windows made the GDI functions available from metafiles, that is, a different context, resulting in the vulnerability. In fact, the exploit does not work on earlier versions of Windows because the code works correctly in the context for which it was originally designed [19].

Another example of the failure of correct code is when the implementation allows circumvention. In 1987 Young, et al. provided the classic example of a provably correct login routine on a successful commercial computer whose memory layout allowed easy circumvention [23].

5.2. Static Analysis

Given a set of known vulnerabilities, we can simply avoid buffer-overflow attacks by fixing or replacing the vulnerable functions. Since this approach works on source code, it is called “Static Analysis.”

Before deploying a program, static analysis can be applied to find and solve the potential problems. To do so, we simply use a predefined knowledge to analyze the source code or binary. For example, knowing that the “strcpy” function in C programming language, a string copy function with no bounds checking, is suspicious to buffer overflow, we can create a profile to search for any use of the function and warn the programmer of the potential threat.

A generic static analysis algorithm is as follows

```
// Static Analysis
1. Open file of signatures
2. Search target source code for signature code
3. if (signature found)
   replace with less vulnerable code
```

DOI:10.4186/ej.2011.15.2.31

5.3. Dynamic Solutions

Since the targets of the attackers are control data, validating the integrity of these data would allow us to detect buffer-overflow attacks. We refer to this class of protection schemes as "Dynamic Solutions", because the systems dynamically manage and verify data in during the execution.

```
// Dynamic Buffer-Overflow Protection
// At run time …

1. Tag control data
2. if (accessing_control_data && legal_tag)
     continue
   else
     raise exception
```

In order to validate (and identify) the integrity of data, it is necessary to have a tag, "metadata" associated with the data. To check the integrity of data, we can simply validate the tag against the data. We differentiate approaches by comparing four components of each dynamic solution:

```
1. Underlying assumptions of the approach
2. Creation of metadata
3. Validation of metadata
4. Handling of invalid data
```

We provide two examples to illustrate these components.

Example 1:
In this first example, buffer overflows are prevented by detecting if control data being used is derived from input (including buffers). In this case, input is being tagged by metadata. When any data is used by control instructions (e.g. branch, return, or call), the system (hardware or software) must verify the metadata to determine that if it is input (by verifying the metadata). In case it is input, an exception is raised.

Example 2:
In this second example, return addresses can only be created by call instructions. When a return address is created, it must be tagged by the call instruction. Other instructions must clear the metadata tag. Return instruction must verify that the address has a valid tag (not modified by other instruction) before executing the return. If the return address is invalid, an exception is raised.

The two examples are different in that their solutions are based on different assumptions. However, they share the same idea that a tag (or metadata) is necessarily attached to critical data. They differ in that the first example tags input in general while the second example specifically tags return addresses.

Dynamic methods can also be differentiated in how the metadata is stored. In general, there are two types of metadata: hardware supported and software managed metadata. Hardware solutions require modification of either hardware organization or instruction set architecture or both. For software managed metadata, the metadata is (mostly) managed by software as normal data. In general, prologue and epilogue are inserted to the program for the creation, validation, and handling of metadata.

Using these components, we can broadly classify dynamic solutions into four subcategories:

- Address Protection
- Input Protection
- Bounds Checking
- Obfuscation

Like Example 2, the address protection schemes assume that addresses (control data) are critical data and must be tagged. Solutions in this scheme modify functions that create the address (e.g. call

instruction) to create the metadata, and modify functions that use the data to verify them. The management of metadata varies among each scheme within this subcategory.

The input protection schemes, such as Example 1, tag input as untrustworthy and invalidate them from being used as internal control data. The main idea is "All input is evil until proven otherwise" [13]. For the hardware solution, hardware provides mechanism for associated metadata with the data (e.g. tagged architecture). If there is an attempt to use input as control data, we can recognize it by validating metadata. The management of metadata is the major difference among them.

Another method is to explicitly validating the boundary of every access to the memory. In this case, a block of data is associated with metadata. This metadata is used to specify base address and limit of each memory reference. We refer to these schemes as bound checking.

Alternatively, it is possible to obscure memory. This would make it difficult to maliciously manipulate the memory though buffer overflow. Given that buffer-overflow attacks can only be done when a memory snapshot is deterministic. The random snapshot would be to attack. Hence, it is extremely hard to write a generic attack script. This class of protection is referred as obfuscation.

5.4. Isolation

Isolation schemes limit the damage from attacks rather than preventing attacks. As a result, their protection is not limited to buffer-overflow attacks. This category includes sandboxing and confinement schemes.

By creating an electronic curtain, isolation schemes can prevent a vector of attack or limit the damage causing by the attack. For example, it is possible to prevent a certain part of memory (e.g. stack) from being used as storage for executable code (i.e. NX). Alternatively, we can jail a compromised process from accessing other parts of memory to contain the damage (i.e. Sanboxing applet).

As with dynamic solutions, isolation can be implemented purely in software or with support from hardware.

5.5. Summary of Protection schemes

In this section, taxonomy for classifying of buffer overflow protection is presented (in Fig. 4). We use three main ideas for the classification: fixing the function, protecting the data, or limiting the effects. In this paper, we refer to these schemes as Static Analysis, Dynamic Solutions, and Isolation respectively. We will continue to review various proposed solutions in the next section.

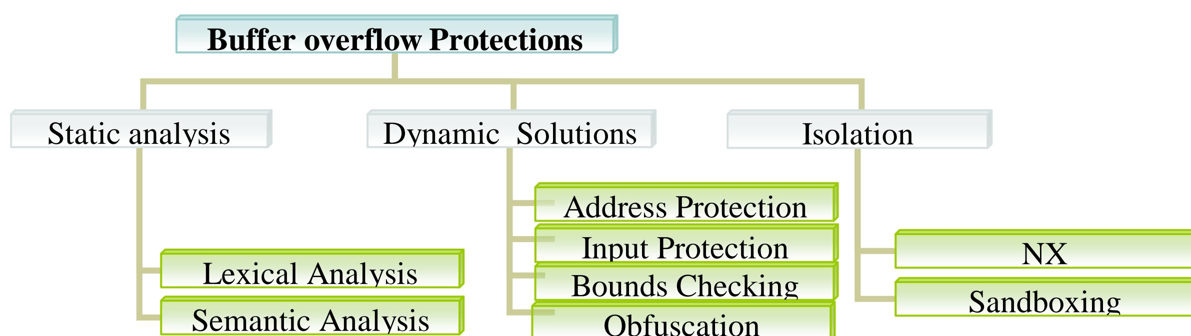

Fig. 4. Taxonomy of solutions against buffer-overflow attacks.

6. Reviews

This section elaborates the proposed buffer-overflow attack solutions.

DOI:10.4186/ej.2011.15.2.31

6.1. Static Analysis

To detect for possible buffer overflow vulnerabilities, tools have been proposed to examine source code. A simple tool is a string search. More sophisticate tools are lexical analyzer and parser. With these tools, programmer can prevent the potential problem before the deployment of software. With no run-time information, all problems may not be detected. Regardless of the false positive, the final solution is still depending on the skill of an individual programmer.

Besides the limitation in detection an overflow in a user-defined function and macro, the main drawback of these tools includes (but is not limited to) false alarms.

6.1.1. Lexical Analysis

Sample tools include "*It's the Software Stupid!*" (ITS4 [25]), FlawFinder [26], RATS [27], STOBO [28], and LibSafe [29].

ITS4 uses lexical analysis and matches tokens in C and C++ against a database of known vulnerabilities. It is useful for highlighting potential security problems as code is written. The report generated from ITS4 includes a short description of the problem, a suggested solution, and the severity.

FlawFinder [26] and RATS [27] are similar tools that scan source code using lexical analysis. Basically, FlawFinder uses the same method as that of ITS4. However, the report generated from FlawFinder is associated with each arguments of each function. A constant string is considered to be safer than a variable string. In addition to C, RATS also supports Perl, PHP, and Python source code.

The Systematic Testing Of Buffer Overflow (STOBO [28]) is an interesting static analysis tool for C programs. It uses profiling to generate its report. This tool will insert special functions into the original program to keep track of each variable and memory allocation. Running the modified program will provide dynamic analysis from the runtime environment.

LibSafe [29] is the implementation of static analysis in the run time environment. It is a safe implementation of a library that is forced to be loaded before the standard C library. Since the library is accessed in the order of loading, Libsafe is able to intercept the known vulnerable functions (e.g. strcpy, strcat).

6.1.2. Semantic Analysis

Sample tools are Splint [30] and BOON [31].

Splint [30] statically checks for possible vulnerabilities in a C program. It is a variation of Lint, a popular static analysis tool for C since the seventies. Splint uses a parser to perform semantic analysis. This means that the tool has a better chance of finding an incorrect use of a function than the lexical analysis counterpart.

BOON [31] is another tool for C programs. It statically checks for security vulnerabilities. In Boon a string is considered as an abstract data type. Thus, it indirectly performs bound-checking analysis. These two ideas provide a framework for the analysis.

6.2. Dynamic Solutions

Several dynamic solutions have been introduced. Each method varies in assumption, choices of metadata and management scheme, or handling routines. We will review them in the following order: Address Protection, Input Protection, Bounds Checking, and Obfuscation. For each method, we will pinpoint the choice of metadata and the management of the metadata.

6.2.1. Address Protection

There are several similar methods that share the assumption of protecting the address. However, they use different types of metadata. These variations include: a canary value, tagged memory, and hardware stack. Though some solutions in this class have become obsolete as attacks have matured, they are a good initial step in preventing buffer-overflow attacks.

6.2.1.1. Canary Words

Assuming that corrupting an address will also corrupt the adjacent data, the validity of address can be verified by validating special adjacent metadata (the canary word)[1]. The general mechanism can be described as:

1. Place a word (canary) adjacent to the address (e.g. return address) on creation (by a call instruction).
2. Verify the canary word before using the address (e.g. before a return instruction uses a return address).

There exist several tools that adopt the concept of canary words. The notable methods are StackGuard [6, 32, 33, 34] and ProPolice [7]. StackGuard assumes that return addresses must not be modified after creation and hence puts a canary word adjacent to the return address. The creation and validation code are injected by a compiler and assume that a buffer overflow occurs only in one direction; ProPolice then reorders the declaration statement to protect against function pointer attacks. Since an overflow only goes in one direction, declaration reordering can prevent the function pointer from being overflowed.

Figure 5 shows the sample of the reordering process. However, this can only protect some variables from being overflowed, but not other variables that are still in the overflow direction.

The main drawback is that buffer-overflow attacks have changed in response to this type of protection to now attack other addresses. In addition, there exists no mechanism for protecting the canary word itself (See [35] for more details).

Original Code	Reorder Code

Int bar() {
    void (* funct2)();
    char buff[80];
    …..
}
	int bar() {
	    char buff[80];
	    void (* funct2)();
	    …..
	}

Fig. 5. Sample of code reordered by IBM ProPolice.

6.2.1.2. Address Encode

Knowing that encryption can help preserving the integrity of data, the same concept is applied to protect the integrity of addresses (e.g. function pointers or return addresses). In this type of protection, the metadata is the key used to encode the data. The general mechanism can be described as:

1. Encode an address with a pre-defined key before storing it to the memory.
2. Decode the address on dereferencing (loading back to the processor).

The most obvious examples of this scheme are PointGuard [36] and Hardware Supported PointGuard [37, 38]. These methods assume that a pointer must not be modified after creation and use a per-process random key for encrypting pointers (generated by software or hardware supported hash table). Besides issues with long-time key management, encrypting a pointer can be problematic with arrays (including C strings) and compile-time assignment. The same issue may also apply to shared libraries and interposes communication.

6.2.1.3. Copy of address

A simple method for preserving the integrity of an address is to preserve another copy of the address. The mechanism can be described as follows:

1. Create a safe copy of the control data (e.g. return address) on creation.
2. Verify the address against the safe copy before using it (e.g. on return).

---

1 A canary word is similar to a canary in a mine: if the canary dies, that indicates a problem.

DOI:10.4186/ej.2011.15.2.31

Several similar tools exist. However, we choose to review a subset including: StackGhost [8], RAS [39, 40, 41], Split Stack [40], SmashGuard [42], RAD Compiler [5], RAD Binary Rewrite [43], DISE [44], StackShield [45], LibVerify [29], and SCACHE [46].

The main differences are in the storage of the redundant copy of the address and the management scheme. For example, StackGhost uses the feature of SPARC processors, register window, for storing a copy; and RAS makes use of the Return Address Stack (RAS), a hardware accelerator for predicting return addresses in some processors. Similarly, RAD, Split Stack, StackShield, Libverify, and SmashGuard allocate a separate stack for storing return addresses. SCACHE uses cache for managing a replica of the return address.

These solutions mainly suffer from the dynamic optimization of control flow (e.g. such as long jump). In addition, they only provide a partial solution—mainly to the return address.

6.2.1.4. Tags

Tagged Architecture [16]

Tagged memories go back at least thirty years with many proposed architectures and a few that made it to market. For our purposes, tagged-memory architectures provide three main functions. The first supports language, and, the Symbolics [15] Lisp computer, which achieved a relatively short-lived commercial success, is a classic example. In that computer, the tags were used to efficiently keep track of dynamic types. The second type uses tags to support capabilities. Capabilities reach back forty years, and their popularity has increased recently with increased interest in security exemplified by capability-based operating systems such as EROS [47]. The tags support the capabilities that control access. The third is best exemplified by the IBM System/38 [48] database computer (the predecessor of the IBM AS/400), which used a tag to protect pointers. Of those, it is the IBM System/38 that is relevant to this survey.

The general idea is to use a special instruction for creating and validating addresses. Though this method sounds promising in that it forces a program to separate addresses from data, it cannot generally apply to legacy code. In other words, a program may have to be modified to use this solution.

6.2.2. Input Protection

Assume that misuse of input as control data can be a potential threat, it is necessary to differentiate them from local data. In this section, three methods that share the same assumption will be reviewed. They are: Minos [20, 49], Tainted Pointer [50], and Secure Bit 2 [51]. Besides the similar assumption, their implementations are fundamentally different. In particular, input is interpreted as data across segments in Minos. For Secure Bit 2, input is data passing from another process (through the kernel). For Tainted Pointer, input is data passed from the operating system.

These methods utilize a tag bit to differentiate between local data and input. However, they differ in the definition of the boundary. For example, Minos uses segmentation to define boundaries. While Tainted Pointer implicitly assumes that the I/O subsystem is a boundary, Secure Bit allows users and operating systems to freely mark anything as input.

This concept sounds promising. However, some may experience problems with multi-threaded programs (e.g. Sun Java JVM) where a group of processes is considered to be one big process and they pass addresses among themselves. Though Tainted Pointer tries to extend this concept to protect pointers in general, it ends up creating a new vector for attacks in that an instruction (e.g. compare) may be needed to untaint data.

DOI:10.4186/ej.2011.15.2.31

6.2.3. Bounds Checking

In Bounds Checking, the methods assume that all accesses to data must be done within the boundary of that variable. We will review two main approaches: software and hardware approaches. For the software, the implementation can be a modification to the compiler of current languages or virtual machine solutions (e.g. type-safe programming languages).

Examples of run-time software solutions are: Array Bounds Checking [52], Rational PurifyPlus [53], and BoundsChecker [54]. Array Bounds Checking [52] is a backward compatible bounds checking in C. Since, the method does not change the representation of a pointer. Thus, it is compatible with the standard C library. For every pointer, a base pointer is defined. A pointer value can be used only for a region. Enforcing bounds checking is simply archived by validating that a memory access is within the specify region defined by the base. Besides being useful, a program may suffer from more than a 30 times slowdown (in a pointer-intensive program). This is directly an overhead of a symbol table. Thus the application of this tool is limited for debugging purpose. A similar case also applies to ***Rational PurifyPlus*** [53], ***BoundsChecker*** [54], ***SafeC*** [24], ***Fail-Safe*** [55], and [56]. Conceptually, we can view this approach as a software implementation of segmentation, which uses an entry in the symbol table as a segment descriptor.

Segmentation is a novel hardware for enforcing bounds checking. Though primarily designed to assist the management of memory relocation, it is used to provide buffer-overflow protection. In the design, only a base register is mandatory for each memory access. IA-32 and I432 [57] associate segmentation with base, limits, and hardware rings. By associate every buffer with base and boundary, a buffer overflow is detected. The drawback is performance degradation and storage required for storing segment descriptor. IA-32 forces that memory accesses must be done though segmentation (in protected mode). However, it can be bypassed by creating one large segment for whole memory. Since segmentation is not common in among architectures, most operating systems bypass segmentation for gain better performance and portability. I432 is another interesting CISC architecture. The main concept of the design is to create a security-aware processor. With ADA programming paradigm, a function is forced to create a new segment as well as a variable has to define data ranges. Giving that I432 instructions ranging from six to 321 bits and each are 10 to 20 times slower than the contemporary VAX 11/780 [58], it was commercially failed.

Similar ideas can also be used with a pointer. For example, ICL 2900 series systems [16], the 1960s architecture, natively supports hardware 'pointer' (a.k.a. descriptor) similar to BoundsChecker. Any dereferences outside the boundary will be detected.

Another approach is to enforce bound checking in run-time environment such as the virtual machine used by type-safe programming languages (e.g. Java Virtual Machine, .Net framework) and interpreted languages. The address bounds checking was accomplished by embedding the metadata and bounds checking mechanism into the virtual machine (run-time environment). Correct bounds checking can prevent buffer overflows. Though the virtual machine can transparently validate storage capacity, the underlying components, standard libraries, and the virtual machine are still C/C++ programs, which are type-unsafe languages. A program created in Type-safe languages can still be vulnerable by buffer overflows. For example, there have been buffer-overflow attacks in Java [26, 4], Perl [59], etc. There also exists a type-safe C: CCured [60].

Bounds checking generally suffer in performance. Unless a segment is created for every variable, it is possible to overflow within a segment. In addition, code with bounds checking tends not to be compatible with legacy code without such a mechanism.

6.2.4. Obfuscation

When there is no appropriate solution, confusion and increased difficulty can be used as a protection mechanism. However, it should not be used alone when other methods are applicable [61]. The most obvious example is Address Obfuscation [62]. Conceptually, this method reorganizes the memory area of each process to make it difficult for attackers. Changing the memory alignment, malicious users will encounter difficulty in overwriting the expected addresses. A compiler is modified to randomly relocate the base addresses of memory segments and the offset between each pair of data items It also permutes the order of variables/routines. A similar mechanism is implemented in the Address Space Layout

DOI:10.4186/ej.2011.15.2.31

Randomization (ASLR) schema of the ***PAX*** project [63], which Microsoft has included in Windows Vista. The ASLR goes further by changing the layout on each reboot. Since the randomness does not occur during execution, a dissembler tool can still reveal the necessary information, but having every installation of an operating system with a different layout helps defeat generic attack tools.

6.3. Isolation

Two concepts are being used for isolation. One is to control the execution of code that may result from buffer-overflow attacks. Another idea is to sandbox the whole process from accessing certain system resources based on a predefined policy. Apart from these two main ideas, variations include the isolation of executable code from being installed or modified during run-time.

Non-Executable Memory: Another common technique is non-executable partitions of memory, such as pages or segments. SPARC processor, as well as several architectures, supports non-executable memory. Recently, AMD (and Intel) added a "NX"[64], a similar feature. The main idea of non-executable memory is to limit code execution to a defined region and. By marking stack, a region for storing local buffer from being executable, a class of buffer-overflow attacks that requires code injection is detected. Microsoft finds NX sufficiently useful to encourage users to enable it. The Solar Designer group [65] and INGO [66] also proposed a patch to the Linux kernel to make a ***non-executable stack***. However, it is still possible to render buffer overflow on return address—leaving the system to be attacked with a valid code in other memory region (code, BSS). Nonetheless, this solution cannot be applied in some cases. For example, a signal handler on Linux (trap) generates code on the stack and executes it on top of stack. Moreover, a functional language, any LISP-like languages, requires an executable stack in their normal operation (a.k.a. trampoline). As a result, only a narrow range of attacks is protected.

SPEF: Alternatively, researchers from Microsoft and the University of California at Los Angeles have developed a Secure Program Execution Framework [67] (SPEF). Instead of protecting the data, the method protects the code. The method makes an installation and an injection of malicious code difficult if not impossible. With SPEF, hardware and compilers are working together to dynamically encrypt and transform code during execution. As a result, install a malware can only be done with special process. However, overflowing the buffer and modifying the control data to a known address is still possible. Based on 3DES and domain ordering hardware, SPEF should experience performance difficulties and may not be feasible for general applications. Another implementation that shares the same concept but uses instruction block signature is [10].

Instruction-Set Randomization: Similar to SPEF, Instruction-Set randomization [9] introduces difficulty in injecting the malicious code. The general idea is to randomize the coding of instructions by XORing them with a key. The authors propose a per-process key schema, which makes it difficult for a dynamic-linked library. As a result, the method supports only static-linked libraries. The technique can also be applied to scripting languages by adding a random number at the end of each instruction and modifying the virtual machine to validate the number in each instruction. In [9], an example of random Perl is presented. The drawbacks are the lack of support for dynamic-linked libraries, the requirement of special hardware, and the limitation of using polymorphic and self-modifying code. Like SPEF, which only prevents the injection of malicious code, overflowing with a known address is possible.

Sandboxing: Sandboxing is a type of policy-enforcement mechanism. Given the malicious nature of buffer-overflow that is a result from information passing from another domain, sandboxing has nothing to do with buffer-overflow attacks. With appropriate policy rules, the damage can be contained. Several levels of sandbox exist: kernel level [68], user level [69, 70, 71], or even hardware-supported sandboxing (e.g. Intel LaGrande [72], TCPA [73,74], TrustZone [75], Microsoft NGSCB [72], ChipLock [76], Bear [73].) In addition, a fine-grained approach to memory management (e.g. MMP [77]) exists. However, only a perfect combination of a security policy and an implementation can prevent buffer-overflow attacks. We believe that it is complementary to other techniques rather than a replacement.

7. Analysis

The ideal solution against buffer-overflow attacks is to create a correct program that is safe from buffer overflow attacks. However, contemporary programming methods cannot delivery this ideal condition at the moment.

Given the definition of buffer overflow and buffer-overflow attacks, the protection schemes will be analyzed by raising issues critical to protections against buffer-overflow attacks. Those issues are: Common Pitfalls, Performance, Compatibility (Transparency), and Deployment and Cost.

7.1. Pitfalls

Buffer-overflow prevention, like many security efforts, has been an "arms-control race" in that attacks have evolved to counter prevention schemes, which in turn require increased sophistication in prevention. In hindsight, we can look back and see why earlier efforts failed. Two themes have emerged:

1. Insufficient assumptions. Some approaches only provide protection against a subset of buffer-overflow attacks. It is likely the maturing of buffer-overflow attacks that have shifted their target. For example, the best known type of buffer-overflow attack modifies return addresses. As soon as developers protected the return address, function-pointer attacks became popular. Another example is trusted code. It can be wrong to assume that trustworthy code is 100% safe from buffer-overflow attacks. Compromising the signed code or the signing mechanism allows execution of malicious 'trusted' code.
2. Insufficient protection of metadata. Metadata is necessary to assist in protecting critical data. Ideally, metadata must not be controllable by attackers. If attackers can control the metadata, they can successfully create buffer-overflow attacks by modifying both data and metadata. For example, if a key or canary can deterministically be reproduced, attacking an encrypted data or guarded canary is possible (More examples in [35, 78,79]).

7.2. Performance

In the tradeoff between performance and security, performance has always received priority. This is best exemplified by segmentation (e.g. I-432 [57]). An appropriate utilization of segmentation is a perfect tool, if one is willing to explicitly declare each variable with a base and limit (including those of integer and floating-point variables, since buffer overflow can also result from type casting). However, the tremendous overhead of the symbol table (or the segment-descriptor table) for such an approach is unacceptable.

7.3. Compatibility (Transparency)

Given a large number of existing programs and libraries, backward compatibility can be a significant requirement. A good product can fail in the marketplace, if it breaks too many things. This paradigm also applies to computers. Here are some compatibility issues critical for solutions against buffer-overflow attacks.

- **Data representation** is a critical problem, if the data representation of the prevention scheme is not interoperable with legacy software libraries. An example is the pointer (and array) representation of PointGuard.
- **Non LIFO control flows** include signal handling, trampoline (a type of software optimization where code is dynamically generating and running on top of the stack frame), and far and near call optimization [21]. In most cases, methods that try to create a redundant copy of data of the stack frame will fail to support the dynamic stack management of non-LIFO control flows. Examples of these methods include: Separated Stack, RAS, RAD, etc.
- **Binary Compatibility** is perhaps the most difficult goal in implementing solutions against buffer-overflow attacks. To support binary compatibility, the solution must maintain not only the same data representation, but it also has to maintain the programming model, the communication protocol (e.g. call and return), and the syntax of every instruction. Most

solutions presented here fail to achieve this goal. However, some code modifications may be necessary in order to provide protection. For example, a minor modification such as a patch to the operating system may provide protection while preserving binary compatibility for all user code. Secure Bit 2, which embedded the mechanism in the hardware, provides binary compatibility for user code. Approaches that use binary rewrite by modifying the program loader also can provide binary compatibility.

7.4. Summary

In Table 1 we show a summary of our analysis of all the protection schemes. This analysis refers to schemes in general. That is, not all schemes map perfectly into our table. However, we believe that this tabulation provides a useful overview of the variety of approaches.

The best solution is secure software developed using state-of-the-art software engineering. In the absence of that, the table indicates that embedding Input Protection into the processor provides the best combination of coverage, penalty and compatibility. Though bounds checking can prevent overflows, it often has a significant performance penalty. When it is possible to rebuild a program and libraries, Address Encoding is promising. However, security is then dependent on another issue: the key-management scheme. When recompiling a program is not possible, Software/Hardware Isolation can provide limited protection.

Given the analysis shown in Table 1, a developer may choose to apply compiler solution (e.g. GCC 4) to assist the detection of address protection. During the software development, a dynamic flow tracking tool (input protection) may ease detecting possible vulnerability. As a last resource, compiler-assisted obfuscation (e.g. GCC 4, MS VC++) and operating systems-assisted obfuscation (e.g. MS Windows Server 2008) would provide minimal protections. Note that hardware-based NX (AMD/Intel NX) can only provide protection against small class of attacks, but it is already included with a new purchased system.

8. Conclusions

When a malformed input is used to overflow a buffer, buffer-overflow attacks occur. This results in a malicious or unexpected behavior of software. Collectively, protection against buffer-overflow attacks is based on an ability to detect and validate control data. Intuitively, from this observation, some metadata is necessary to distinguish between input and local data.

Among the proposed solutions, there are only three themes: preventing **functions** from overflowing data, protecting target data or tagging **data** that may cause buffer overflow, and limiting the **effects** caused by buffer-overflow attacks. These themes are embedded in Static Analysis, Dynamic Solutions, and Isolation respectively.

While we have to rely mostly on programmers' skill, the most practical solutions today are solutions that based on address protection scheme and obfuscation scheme. This is due to the fact that they can be applied easily without the modification of underlying architecture. (i.e. programs, compiled with a special compiler, are less vulnerable to buffer-overflow attacks.) This direction is being pursued by the software industries.

In the big picture, every solution that protects against buffer-overflow attacks comes at some cost. Also, as attacks have matured some solutions have proven insufficient for providing full protection against all classes of attacks. . In particular, provably correct code should eliminate buffer overflows (as well as most other errors). However, currently these techniques are not widely applied, so vulnerable code persists. We conclude that no solution except correct code is perfect in every aspect.

DOI:10.4186/ej.2011.15.2.31

Table 1. Comparison of Protection schemes.

	Return Address (*Y/N/Likely*)	Function Pointer* (*Y/N/Likely*)	Coverage (*Lo, Mid, High*)	Mechanism (*HW/Compiler, VM*)	Programmers Involvement (*Lo, Mid, High*)	Transparent to Legacy Code (*Y/N*)	Performance Penalty (*Lo, Mid, High*)
Static Analysis Address Protection	Y	Y	Mid	Compiler	High	N	Lo
Canary Word	Y	N	Mid	Compiler	Lo	N	Lo
Address Encode	Y	Y	High	Compiler	Lo	N	Lo
Copy of Address	Y	N	Mid	Compiler	Lo	N	Lo
Tag	Y	Y	Mid	HW/Compiler	Lo	N	Lo
Input Protection	Y	Y	High	HW	Lo	Y	Lo
Bounds Checking							
Segmentation	Y	Y	High	HW/Compiler	Mid	N	Mid
Run-time Extension	Y	Y	High	Compiler	Mid	N	High
type-safe	Y	Y	High	VM	Mid	N	High
Obfuscation	Y	Y	Lo	Compiler	Lo		Mid
Isolation							
NX	Likely	Likely	Mid	HW	Lo	Y	Lo
HW Sandbox	Likely	Likely	Mid	HW	Lo	Y	Mid
SW Sandbox	Likely	Likely	High	Compiler/VM	Mid	Y	High
Instrument	Likely	N	Lo	HW/VM	High	N	High

*Function pointers also include global-offset table.

8.1. Future research

Research in buffer overflows are now heading in two directions: security-enhanced programming languages (including debugging tools) and dynamic run-time environment that can transparently detect and prevent systems from possible vulnerabilities. For the first direction, researchers aim at enhancing programming language and development environment to avoid buffer-overflow vulnerabilities in the first place (i.e. static analysis). These solutions include a patch to a compiler to perform analysis during code compilation. Another direction aims at preventing buffer-overflow attacks without modifying binaries. Both directions are being researched heavily. Nonetheless, they are not mature enough to completely render buffer-overflow attacks impossible. Given the history of buffer-overflow attacks, we believe that a practical solution to buffer-overflow vulnerability is still an open topic.

References

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

[2] CERT: Computer Emergency Response Team at Carnegie Mellon University. Available: http://www.cert.org

[3] D. Dean, E. W. Felten, and D. S. Wallach, "Java security: from HotJava to Netscape and beyond," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1996, pp. 190-200.

DOI:10.4186/ej.2011.15.2.31

[4] Sun Alert Notification. (2004). *Document ID 57643: Netscape NSS library vulnerability affects Sun Java Enterprise System*.
[5] T. Chiueh and F. H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *International Conference on Distributed Computing Systems*, 2001, pp. 409-417.
[6] C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting systems from stack smashing attacks with StackGuard," presented at the Linux Expo, Raleigh, NC, 1999.
[7] J. Etoh. (2000). *GCC Extension for protecting applications from stack-smashing attacks*. IBM.
[8] M. Frantzen and M. Shuey, "StackGhost: hardware facilitated stack protection," in *Proceedings of the 10th USENIX Security Symposium*, 2000.
[9] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM Conference on Computer and Communication Security*, 2003, pp. 272-280.
[10] M. Milenković, A. Milenković, and E. Jovanov, "Using instruction block signatures to counter code injection attacks," *SIGARCH Computer Architecture News,* vol. 33, pp. 108-117, 2004.
[11] Webopedia Computer Dictionary. *What is buffer overflow?* Available: http://www.webopedia.com/TERM/B/buffer-overflow.html
[12] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *Proceedings of the 10th USENIX Security Symposium*, 2001.
[13] M. Howard and D. Leblanc, "Chapter 10: all input is evil!," in *Writing Source Code*, 2nd ed: Microsoft Press, 1965.
[14] A. Glew, "Segments, capabilities, and buffer overrun attacks," *ACM SIGARCH Computer Architecture News,* vol. 31, pp. 26-31, Sep. 2003.
[15] D. A. Moon, "Symbolics architecture," *Computer,* vol. 20, pp. 43-52, 1987.
[16] E. F. Gehringer and J. L. Keedy, "Tagged architecture: how compelling are its advantages?," in *International Symposium on Computer Architecture*, 1985, pp. 162-170.
[17] E. Chien and P. Ször, "Blended attacks exploits, vulnerabilities and buffer-overflow techniques in computer virus," in *Proceedings of Virus Bulletin Conference*, 2002.
[18] A. One. (1996) Smashing stack for fun and benefit. *Phrack Magazine*.
[19] M. Howard and S. Lipner, *The Security Development Lifecycle*: Microsoft Press, 2006.
[20] J. R. Crandall and F. T. Chong, "Minos: control data attack prevention orthogonal to memory model," in *International Symposium on Microarchitecture*, 2004, pp. 221-232.
[21] K. Piromsopa and R. Enbody, "Buffer Overflow: Fundamental," Michigan State University Technical Reports #MSU-SE-04-47, 2004.
[22] BLEXIM. (2002). *Basic Integer Overflow*. Available: http://www.phrack.org/phrack/60/p60-0x0a.txt
[23] W. D. Young and J. McHugh, "Coding for a believable specification to implementation mapping," presented at the IEEE Symposium on Security and Privacy, 1987.
[24] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994, pp. 290-301.
[25] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "ITS4: a static vulnerability scanner for C and C++ code," in *Proceedings of the 16the Annual Computer Security Applications Conference*, 2000.
[26] Flawfinder. Available: http://www.dwheeler.com/flawfinder/
[27] RATS. Available: http://www.securesw.com/rats/
[28] E. Haugh and M. Bishop, "Testing C programs for buffer overflow vulnerabilities," in *Proceedings of the 2003 Symposium on Networked and Distributed System Security (SNDSS 2003)*, 2003.
[29] A. Baratloo, N. Singh, and T. Tsai, "Transparent run-time defense against stack smashing attacks," in *Proceedings of the USENIX Annual Technical Conference*, 2000.
[30] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software,* vol. 19, pp. 42-51, 2002.
[31] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *Proceedings of the 10th USE NIX Security System*, 2001.

DOI:10.4186/ej.2011.15.2.31

[32] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks," in *the Proceedings of the 7th USENIX Security Symposium*, 1998.
[33] C. Cowan, P. Wagle, C. Pu, S. Beattie, and W. J., "Buffer overflows: attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Expo (DISCEX)*, 2000.
[34] H. Hinton, C. Cowan, L. Delcambre, and S. Bowers, "SAM: Security Adaptation Manager," in *Proceedings of the Annual Security Applications Conference (ACSAC)*, 1999.
[35] Bulba and Kil3r. (2002) Bypassing stackguard and stackshield. *Phrack Magazine*.
[36] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th USENIX Security Symposium*, 2003.
[37] Z. Shao, Q. Zhuge, Y. He, and E. H.-M. Sha, "Defending embedded systems against buffer overflow via hardware/software," in *Proceedings of the 20th Annual Computer Security Applications Conference*, Tucson, Arizona, Dec. 6-10, 2004.
[38] N. Tuck, B. Calder, and G. Varghese, "Hardware and binary modification support for code pointer protection from buffer overflow," in *Proceedings of the 37th International Symposium on Microarchitecture*, 2004, pp. 209-220.
[39] J. McGregor, D. Karig, Z. Shi, and R. B. Lee, "A processor architecture defense against buffer overflow attacks," in *Proceedings of the IEEE International Conference on Information Technology.: Research Education (ITRE 2003)*, 2003, pp. 243-250.
[40] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, "Architecture support for defending against buffer overflow attacks," in *Workshop on Evaluating and Architecting Systems for Dependability*, 2002.
[41] D. Ye and D. Kaeli, "A reliable return address stack: microarchitectural features to defeat stack smashing," *ACM SIGARCH Computer Architecture News,* vol. 33, 2005.
[42] H. Ozdoganoglu, T. N. Vijaykumar, C. E. Brodley, A. Jalote, and B. A. Kuperman, "SmashGuard: A hardware solution to prevent security attacks on the function return address," Department of Electrical and Computer Engineering, Purdue University Tech Report (TR-ECE 03-13), 2003.
[43] M. Prasad and T. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks," in *Usenix Annual Technical Conference, General Track*, 2003.
[44] M. L. Corliss, E. C. Lewis, and A. Roth, "Using DISE to protect return addresses from attack," *ACM SIGARCH Computer Architecture News,* vol. 33, 2005.
[45] VENDICATOR. (2000). *Stack Shield technical info file v0.7*.
[46] K. Inoue, "Energy-security tradeoff in a secure cache architecture against buffer overflow attacks," *ACM SIGARCH Computer Architecture News,* vol. 33, 2005.
[47] J. S. Shapiro, "EROS: a capability system," Department of Computer and Information Science, University of Pennsylvania Technical Report MS-CIS-97-04.
[48] S. H. Dahlby, G. G. Henry, D. N. Reynolds, and P. T. Taylor, "Chapter 32. the IBM System/38: a high-level machine," in *Computer Structures: Principles and Examples*, ed, 1982.
[49] J. R. Crandall and F. T. Chong, "A security assessment of the Minos architecture," *ACM SIGARCH Computer Architecture News,* vol. 33, 2005.
[50] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer, "Defeating memory corruption attacks via pointer taintedness detection," in *Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN)*, Yokohama, Japan, Jun. 28-Jul. 1, 2005.
[51] K. Piromsopa and R. Enbody, "Secure Bit2 : transparent, hardware buffer-overflow protection," Department of Computer Science and Engineering, Michigan State University Technical Reports #MSU-CSE-05-9, 2005.
[52] R. W. M. Jones and P. H. J. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," in *the 3rd International Workshop on Automated Debugging*, 1997.
[53] Rational PurifyPlus. IBM Rational Software.
[54] BoundsChecker. Available: http://www.compuware.com/products/devpartner/bounds.htm
[55] Y. Oiwa, T. Sekiguchi, E. Sumii, and A. Yonezawa, "Fail-safe ANSI-C compiler: an approach to making C programs secure progress report," vol. 2609, ed, 2003, pp. 133-153.

[56] K. Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities," *Software--Practice & Experience* vol. 33, pp. 423-460, 2003.
[57] E. Organick, *A Programmer's View of the Intel 432 System*: McGraw-Hill, 1983.
[58] R. P. Colwell, C. Y. Hitchcock, E. D. Jensen, H. M. B. Sprunt, and C. P. Kollar, "Instruction sets and beyond: computers, complexity and controversy," *Computer,* vol. 18, pp. 8-19, 1985.
[59] U.S. Department of Energy Computer Incident Advisory Capability. (2004). *O-130: Perl and ActivePerl win32_stat Buffer Overflow*. Available: http://www.ciac.org/ciac/bulletins/o-130.shtml
[60] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *the Proceedings of the Principles of Programming Languages*, 2002, pp. 128-139.
[61] J. C. Cannon, *Privacy: What Developers and IT Professionals Should Know*: Addison Wesley Professional, 2004.
[62] S. Bhatkar, D. C. Duvarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *Proceedings of the 12th USENIX Security Symposium*, 2003.
[63] PAX Team. (2003). *Documentation for the PAX project*.
[64] T. Krazit. (2004) AMD chips guard against Trojan horses. *PCWorld News*.
[65] Solar Designer. (2002). *Linux kernel patch from the Openwall Project (Non-Executable User Stack)*. Available: http://www.openwall.com
[66] Ingo. (2004). *Exec Shield, new Linux security feature*.
[67] D. Kirovski, M. Drinić, and M. Potkonjak, "Enabling trusted software integrity," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 108-120.
[68] D. S. Peterson, M. Bishop, and R. Pandey, "Flexible containment mechanism for executing untrusted code," in *Proceedings of the 11th USENIX UNIX Security Symposium*, 2002.
[69] F. Chang, A. Itzkovitz, and V. Karamcheti, "User-level resource-constrained sandboxing," in *Proceedings of the 11th USENIX UNIX Security Symposium*, 2000.
[70] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *Operating Systems Review (ACM),* vol. 27, pp. 203-216, 1993.
[71] C. Small, "MiSFIT: a tool for constructing safe extensible C++ systems," in *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, Portland, Oregon, Jun. 1997.
[72] Microsoft Corporation. (2004). *The next-generation secure computing base: an overview*.
[73] R. Macdonald, S. W. Smith, J. Marchesini, and O. Wild, "Bear: an open-source virtual secure coprocessor based on TCPA," Department of Computer Science, Dartmouth College Technical Report TR2003-471, 2003.
[74] Trusted Computing Platform Alliance. (2004). *TCPA IT White Paper*.
[75] ARM. (2004). *TrustZone Technology*.
[76] T. Kgil, L. Falk, and T. Mudge, "ChipLock: support for secure microarchitectures," *ACM SIGARCH Computer Architecture News,* vol. 33, 2005.
[77] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *ASPLOS-X*, Oct. 2002, pp. 304-316.
[78] D. Litchfield. (2003). *Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 Server*. NGS Software.
[79] T. Newsham. (1997). *BugTraq: Re: StackGuard: automatic protection from stack-smashing attacks*.