Arbitrary Copy: Bypassing Buffer-Overflow Protections

Krerk Piromsopa, *Member, IEEE*, and Richard J. Enbody, *Member, IEEE*

Abstract— **Recent advances in buffer-overflow protection are able to eliminate several common types of buffer-overflow attacks (e.g. stack smashing, jump table). In this paper, we introduce arbitrary copy, a type of buffer-overflow attack that is capable of bypassing most buffer-overflow solutions. By overflowing both source and destination pointers of any string copy (or similar) function, arbitrary copy is able to utilize a useful local address for attacking a system. This method can bypass even the most promising buffer-overflow protection that enforces the integrity of address such as Secure Bit [24] and MINOS [8]. Later, we analyze conditions necessary for the success of this attack. Though satisfying all necessary conditions for this attack should be difficult, our conclusion is that it is a potential threat and requires consideration.**

Index Terms—**Buffer overflow, Buffer-Overflow Attacks, Computer security, Intrusion Detection, Intrusion Prevention**

I. Introduction

In this paper, we will present a type of buffer-overflow attack that is able to bypass most buffer-overflow protections. We refer to this attack as "***arbitrary copy***". Arbitrary copy is an attack on two data pointers. The successful attack allows an attacker to copy data from one location to another arbitrarily.

Although, they date back to the infamous MORRIS worm of 1988 [26], buffer-overflow attacks remain the most common. Though skilled programmers should write code without buffer overflows, no program is guaranteed free from bugs so it cannot be considered completely secure against buffer-overflow attacks. The persistence of buffer-overflow vulnerabilities speaks to the difficulty of eliminating them. In addition, as buffer overflow vulnerabilities are eliminated in operating systems, they are being found and exploited in applications. When applications are run with root or administrator privileges the impact of a buffer overflow is equally devastating.

In an effort to avoid relying on individual programming skill, a number of researchers have proposed a variety of methods to protect systems from buffer-overflow attacks. Most of them are not able to provide complete protection. For example, some only prevent the original stack-smashing attack, but can be circumvented by more recent attacks.

The goal of this paper is to provide a rudimentary understanding of arbitrary copy attacks. We begin with background of buffer-overflow attacks and current protection schemes. Next, we examine the arbitrary copy and its potential threat. Later is the analysis of a possible solution.

K. Piromsopa is a Ph.D. student in the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA (corresponding author to provide phone: 517-353-3148; fax: 517-432-1061; e-mail: piromsop@cse.msu.edu).

R. J. Enbody is with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA. (e-mail: enbody@cse.msu.edu).

II. Background

This section begins by reviewing the characteristics of buffer-overflow vulnerabilities and attacks. Later we briefly analyze current solutions against buffer-overflow attacks. In particular, we will focus on a promising approach, namely ***input protection***.

A. Buffer-Overflow Attacks

Buffer-overflow attacks occur when a malformed input is being used to overflow a buffer causing a malicious or unexpected result. Some metadata is necessary for prevention [13].

There are two main targets of buffer-overflow attacks: control data and local variables. In the vast majority of attacks, control data is the target so prevention schemes have focused on control data. Control data can be divided into several types: return addresses, function pointers, and branch slots. Return addresses have been the primary target since their location can easily be guessed. More advanced buffer-overflow attacks target other control data. Some literature refers to attacks on return addresses as first-generation attacks, and those on function pointers as second-generation attacks [3].

B. Current Protection Schemes

Current approaches against buffer-overflow attacks can be partitioned into three broad categories: static analysis, dynamic solutions, and isolation. Static analysis tries to fix functions that are vulnerable to buffer-overflow attacks. Dynamic approaches monitor or protect data that is either a target or the source of buffer-overflow attacks. Isolation seeks to limit the damage of attacks.

The main idea of "Static analysis" is to find and solve the problem before deploying the program. To do so, we first analyze the source code or disassembly of the program by looking for code with a predefined signature. Examples of

tools in this category are: ITS4 [30], FlawFinder [11], RATS [25], and STOBO [14]

Knowing which data are critical to attacks, we can prevent attacks by validating the integrity of that data. As mentioned above, the data of interest are control data such as (but not limited to) return addresses. We name these "Dynamic Solutions" because data are dynamically managed and verified in the run-time environment. We will further elaborate the tools in this category later.

Isolation schemes isolate the attacker either to eliminate an attack vector or to contain damage after a successful attack. Preventing the execution of code in stack memory isolates the stack from the attacker. Alternatively, limiting the memory of a process can isolate a compromised process. NX nonexecutable memory is an example of the former while sandboxing is an example of the latter. Examples include AMD NX [19], non-executable stack [28], SPEF [18], and sandboxing.

In a survey of buffer-overflow protection [24], it is suggested that metadata is necessary for validating the integrity of data. While the assumptions of critical data and the methods for storing and validating metadata vary from one solution to another, dynamic solutions can be classified into four groups:

- Address Protection
- Input Protection
- Bounds Checking
- Obfuscation

The address protection schemes share the assumption that addresses (e.g. return address) are critical data and must be tagged. In these schemes the metadata is created by functions that create the address (e.g. call instruction), and verified by the many instructions that use the address (e.g. return instruction). The schemes within this group are differentiated by the types of metadata they use. Examples are StackGuard [6], ProPolice [10], PointGuard [7], Hardware Supported PointGuard [27], StackGhost [12], RAS [20], RAD [4], DISE [5], SCACHE [16].

The input protection schemes are latest and most promising. These schemes assume that external data are untrustworthy and should not be used as internal control data. The underlining concept is that "All input is evil until proven otherwise" [15]. In most cases, metadata are tightly coupled to the data in hardware (e.g. tagged memory). Data from external sources are tagged so it can be recognized, if there is an attempt to use it as control data. Implementing the metadata in hardware makes attacking the protocol difficult—maybe impossible. The schemes in this group differ in the management of metadata. In the next section, we will focus on this approach.

Rather than tagging data, bounds checking schemes explicitly bound buffers to prevent overflow. In this case, the metadata is associated with every block of allocated data and is used to bound accesses. The notable tools are Array Bounds Checking [17], Segmentation, and type-safe programming languages.

Instead of protecting the data directly, obfuscation schemes reorganize memory to obscure memory making malicious manipulation of memory through buffer overflows more difficult. These schemes assume that attackers rely on a certain snapshot of addresses to overflow the critical data. If the snapshot is random or difficult to guess, an attack is more difficult. Address Obfuscation [1] and ASLR [22] are good examples.

Taxonomy and more details of buffer-overflow protection schemes can be found in a survey of buffer-overflow protection [24].

C. *Input Protection*

Input protection schemes are dynamic solutions against buffer-overflow attacks. The underlying assumption is that input data should be treated differently from local data, and should not be used as control data. We will review four methods: Minos [8] and [9], Tainted Pointer [2], Dynamic Flow Tracking [29], Dynamic Taint Analysis [21], and Secure Bit [23] that share the same assumption, but different implementations. Minos views data across segments as input. Tainted Pointer considers data passed from the operating system as input. Dynamic Flow Tracking relies on operating systems for marking input. Secure Bit treats data passing between processes through the kernel as input.

In addition to addresses, Tainted Pointer also tried to prevent input from being used as a pointer. However, input is sometimes used as a part of pointer arithmetic (e.g. indexing). This aspect of Tainted Pointer may raise false alarms in many programs.

The other schemes protect a process from external control data, but do not prevent buffer-overflow attacks on non-control data. That raises the question: can an attacker use a buffer-overflow attack on non-control data to manipulate local control data to modify control flow?

III. ARBITRARY COPY

There exists an arbitrary copy primitive which may allow attackers to modify control flow without using external control data. Using strcpy one can construct a vulnerable routine such that using a buffer-overflow to modify source and destination pointers, an attacker can arbitrarily copy any data from one location to another. This technique allows an existing piece of control data, an address with no Secure Bit set, to overwrite another piece of control data. The result is control flow other than what the original programmer intended. Necessary conditions for the success of this type of attack are:

1. A vulnerable copy function such that a user can modify both arguments (source and destination pointers) (possibly using buffer-overflow attacks) as exemplified in Figure 1.
2. The (useful) control data is stored in the local memory area.

```
char *src,*dest;
char buff[10];

gets(buff);
…
strcpy(src,dest);
```

Figure 1 Vulnerable code

Both of these conditions must be true. If one fails, the attack fails. Though the first condition could be satisfied in any arbitrarily program, the code generated by the compiler will likely render the attack impossible. For example, any level of optimization will use registers for storing the source and destination variables. If either or both are in registers, a buffer-overflow to modify both variables will fail. We will analyze the possible cases where both conditions concurrently occur later.

A. Example

To ease understanding, Figure 2 presents a sample case of an attack on non-control data where the vulnerability might be applicable.

```
int b() {
  char *src,*dest;
  char buff[10];
  printf("Input string:.\n");
// Overflow *src, *dest
  gets(buff);
// Copy src to dest
  strcpy(src,dest);
}

int a() {
  …
  b();
  …
}

int main (int argc,char *argv[]) {
  a();
}
```

Figure 2 Sample Buffer-Overflow attacks on non-control data

In this example, main calls function "a" which then calls the vulnerable function "b". Within "b" the user inputs `buff` which can overflow to both overwrite `*src` to point to the return address of a previous call (e.g. "`a()`") and overwrite `*dest` to point to the target address (e.g. return address of "`b()`" or "`main()`"). Note that this overflow is possible only if all optimization is turned off so that neither `src` nor `dest` is in a register. Under these circumstances it is possible to change the control flow without replacing control data with external data—only internal data is used. Note that the damage in this example is to create an infinite loop or crash the program, effectively a denial of service to the process.

While most internal data targets will be benign, one can imagine malicious possibilities, even if they are a bit far-fetched. For example, if for some reason a programmer created a function pointer to shell and had both a vulnerable copy routine and no optimization; one could copy that shell pointer elsewhere to allow a shell call someplace different than the programmer intended. Note that the desired *privileged-elevated* shell is not possible with this attack because the best buffer-overflow prevention schemes will prevent privilege-elevation attacks. Alternatively, (again with a vulnerable copy routine and no optimization) if one had function pointers to both an authorization "accept" function and a "reject" function one might be able to redirect program flow to subvert an authorization routine to the "accept" function when the "reject" function was expected.

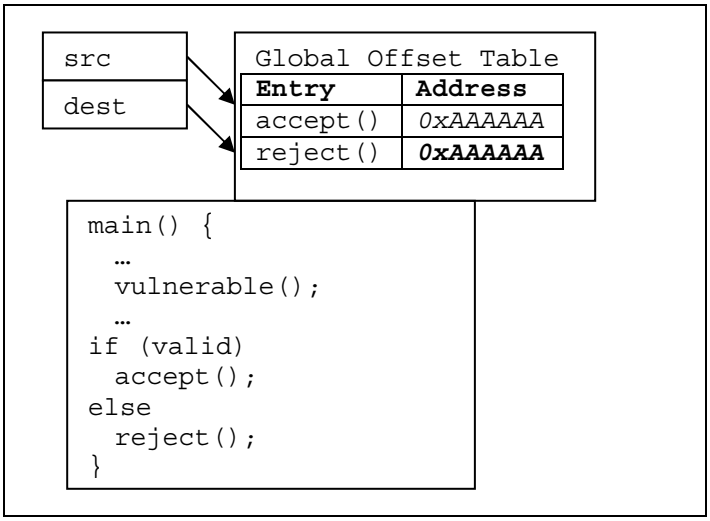

Figure 3 a possible scenario

IV. ISSUES

Consider the second condition of arbitrary copy: the presence of a useful address in local memory. We know that a mechanism like Secure Bit prevents the use of input as control data, thus only purely local data that is not derived from input is a potential threat. One's first thought might be that any function call could provide an address of that function. However, because of relocation, local calls use relative addresses which cannot be used for this attack. Other sources of control targets such as jumps are also relative addresses and not useful. Given this observation, potential sources of addresses are narrowed to the presence of a shared library or a function pointer.

Shared library. In the case of shared libraries (the function is located in the shared library), a call to the function means there exists a useful entry in the Global Offset Table (GOT).

Function Pointer. The assignment of a function address to

a function pointer (frequently found in C++) would create a pointer available for reuse.

If a useful address is stored as an entry in the GOT or a function pointer, the buffer-overflow described above can be used to replace a target address with this address. Target addresses might be return addresses, function pointers, or an entry in GOT itself.

The probability that all conditions are applicable is considered to be low. In fact, some researchers [8] do not believe that it will be a problem or suggest that encoding addresses in GOT should be sufficient for preventing the attack. However, that prevention might not be able to protect some function pointers in C++.

V. Possible Solutions

Though the attack sounds probabilistically low, it is not impossible, and experience suggests that no matter how remote the possibility, someone, sometime will exploit it. We have already mentioned that the most simplistic optimization prevents the attack. To protect against this attack in the presence of no optimization, we simply have to eliminate at least one critical condition. There are three possible methods.

- Prevent a raw address from being stored directly in the program.
- Secure the target address from being modified (e.g. GOT and function pointers).
- Validate that both the source and destination pointer have not been maliciously modified.

Rather than storing an address directly into the GOT table or function pointer, we may choose to store an encoded version of an address or store a relative address. Even a trivial encoding such as XOR (like PointGuard [7]) with some constant would be sufficient. However, this approach does not prevent a copy between locations that share the same encoding scheme or key used to encrypt the address (e.g. between function pointers or entries in the GOT). Note that PointGuard [7] can be used to reduce the probability of overwriting source and destination pointers. However, if the key and algorithm can be circumvented, it is possible to overwrite it with a valid copy. In fact, we may be able to overflow the value (e.g. index) that is used for pointer arithmetic rather than modifying the pointer directly.

Rather than making the useful address useless, we can protect the target from being modified. In the case of GOT, we can protect the GOT from being a target by declaring it as read-only after the shared library is configured. Nonetheless, we cannot apply the same idea to protect function pointers or return addresses in general.

Alternatively, we can validate (assert) the source and destination pointers before running the "strcpy(..)" function. If the source and destination pointers can be validated, the attack can be prevented. However, a false alarm may be generated when a pointer is the arithmetic result of input.

VI. Conclusions

Arbitrary copy is a potential threat that can bypass current state-of-the-art buffer-overflow protection schemes. While other (easier) vectors of attacks still exist, it is unlikely that arbitrary copy will be used as a tool. However, the recent advances in buffer-overflow protection will make existing attacks obsolete. While trivial optimization eliminates the threat, one cannot count on non-optimization as a complete solution. We should pay a close attention to this problem.

We are now working on extending Secure Bit to protect against buffer-overflows of non-control data. In addition to the broader protection provided, this specific attack can be prevented by preserving the integrity of the source and destination pointers from illegal modification. The common practice of using user input as array indices complicates identifying illegal modification.

.

References

[1] S. Bhatkar, D. C. Duvarney, and R. Sekar, "Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits," In *Proc. of the 12th USENIX Security Symposium*. 2003.

[2] S. Chen, J. Xu, N. Nakka, Z. KalbarcZyk, and R. K. Iyer, "Defeating Memory Corruption Attacks via Pointer Taintedness Detection," in *Proc. Of IEEE International Conf. on Dependable Systems and Networks (DSN)*, Yokohama, Japan, June 28 - July 1, 2005

[3] E. Chien, and P. Szor, "Blended Attacks Exploits, Vulnerabilities and Buffer-Overflow Techniques in Computer Viruses," In *Proc. of Virus Bulletin Conf*, 2002

[4] T. Chiueh, F. Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks," In *Intl. Conf. on Distributed Computing Systems*, 2001.

[5] M.L. Corliss, E.C. Lewis, and A. Roth, "Using DISE to Protect Return Addresses from Attack," *ACM SIGARCH, Vol 33. No. 1,* 2005.

[6] C. Cowan, S. Beattie, R.F. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting Systems from Stack Smashing Attacks with StackGuard," the Linux Expo, Raleigh, NC,1999

[7] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities," In *Proc. of the 12th USENIX Security Symposium*, 2003

[8] J.R. Crandall, and F.T. Chong, "Minos: Control Data Attack Prevention Orthogonal to Memory Model," *Intl. Sym. on Microarchitecture*, 2004.

[9] J.R. Crandall, and F.T. Chong, "A Security Assessment of the Minos Architecture," *ACM SIGARCH, Vol 33. No. 1*, 2005.

[10] J. Etoh, "GCC extension for protecting applications from stack-smashing attacks," IBM, 2000.

[11] Flawfinder, Available: http://www.dwheeler.com/flawfinder/

[12] M.S. Frantzen, "StackGhost: Hardware facilitated stack protection," *In Proc. of the 10th USENIX Security Symposium*, 2000.

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

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

[15] M. Howard, D. Leblance, Chapter 10:All Input Is Evil!. Writing Secure Code, Microsoft Press, 2nd ed.(1965)

[16] K. Inoue, "Energy-Security Tradeoff in a Secure Cache Architecture Against Buffer Overflow Attacks," *ACM SIGARCH, Vol 33. No. 1*, 2005.

[17] R.W.M. Jones, and P.H.J. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," In The *3rd Intl. Workshop on Automated Debugging*, 1997.

[18] D. Kirovski, M. Drinic, and M. Potkonjak, "Enabling Trusted Software Integrity," ACM *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[19] T. Krazit, "PCWorld - News - AMD Chips Guard Against Trojan Horses," IDG News Service, 2004.

[20] J.P. Mcgregor, D.K. Karig, Z. Shi, R.B. Lee, "A Processor Architecture Defense against Buffer Overflow Attacks," In *Proc. of the IEEE Intl. Conf. on Information Tech.: Research and Education (ITRE 2003)*, 243-250.

[21] J. Newsome, and D. Song, "Dynamic Taint Analysis: Automatic Detection and Generation of Software Exploit Attacks," In *NDSS* (Feb, 2005)

[22] PAX TEAM. 2003. Documentation for the PaX project

[23] K. Piromsopa, and R. Enbody, "Secure Bit2 : Transparent, Hardware Buffer-Overflow Protection," Technical Reports #MSU-CSE-05-9, Department of Computer Science and Engineering, Michigan State University, 2005.

[24] K. Piromsopa, and R. Enbody, "Survey of Buffer-Overflow Protection," Technical Reports #MSU-CSE-06-3, , Department of Computer Science and Engineering, Michigan State University, 2006.

[25] RATS, Available: http://www.securesw.com/rats/

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

[27] Z. Shao, Q. Zhuge, Y. He, and E.H. Sha, "Defending Embedded Systems Against Buffer Overflow via Hardware/Software," In *Proc. of the 20th Annual Computer Security Applications Conference*, Tucson, Arizona (Dec. 6-10, 2004)

[28] SOLAR DESIGNER, Linux kernel patch from the Openwall Project (Non-Executable User Stack), 2002. Available: http://www.openwall.com/

[29] G. Suh, J. Lee, and S. Devadas, "Secure program execution via dynamic information flow tracking," In *ASPLOS XI* (Oct, 2004.)

[30] J. Viega, J.T. Bloch, Y. Kohno, and G. Mcgraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code," In *Proc. of the 16th Annual Computer Security Applications Conferenc*e, 2000.