Defending Against the Buffer Overflows: A Static Approach Using Proof-Carrying Code

Lei Wang and Ping Wang
School of Computer Science and Engineering, Beihang University, Beijing, China
Email: wangle@buaa.edu.cn, wangping2012@cse.buaa.edu.cn

Abstract—Buffer overflow has been one of the most outstanding attacks in the last ten years. This kind of vulnerability may compromise the system security by various means. Existing solutions to this problem have focused on the execution environment of the malicious program rather than the hypothesis of buffer overflow and most of them try to detect buffer overflows dynamically. This paper presents the effort of applying a static analysis approach against the programs exploiting buffer overflow and the method adopted is named Proof-Carrying Code (PCC). This paper shows that: (1) it is possible to defend against most of the buffer overflow vulnerabilities with proper use of PCC; and (2) the method is well prepared to handle the coming-up variants of the buffer overflow problems.

Index Terms—proof-carrying code, buffer overflow, static analysis

I. INTRODUCTION

Buffer overflows have been the most common form of security vulnerability in the last ten years and this kind of vulnerability has been exploited by various means to attack the system security. For instance, the original Morris Worm used a buffer overflow against the fingerd program to corrupt the name of a file that fingerd would execute and this may be one of the most outstanding attacks based on buffer overflows [1], [2]. Moreover, this kind of vulnerability has been found on extensive spread over the Internet.

Buffer overflow vulnerabilities and attacks come in various forms, as well as the protection solutions to the existing problems. The buffer may locate on the stack, on the heap or in the static data area and the buffer overflow programs may seek to corrupt the activation records, the function pointers or the longjmp buffers, etc [3], [4]. Existing protection solutions to the buffer overflow problems have focused on the execution environment of the vulnerable programs and most of them struggle to patch the security hole by improving the execution environment rather than eliminate the problem itself. The difficulty of the existing solutions result from the fact that what the host system gets from the code producer is merely the binary code, with which static analysis is almost impossible. Therefore the only approach the host may take is to get the binary code executed and hope to detect the security hole dynamically. Actually, the programs with buffer overflow vulnerabilities have been following some conventional formats so that it is a trivial task to find them out provided we have the source code in hand. This is exactly where Proof-Carrying Code [5] finds its place. A contract is established between the host system and the code producer by the introduction of the certification process based on the safety policy published by the host system. Thus it becomes feasible to detect the buffer overflow vulnerabilities through static analysis on the source code.

We present a brief retrospection on some efficient solutions to the buffer overflow problems in Section 2, including Libsafe and PaX. In Section 3, we introduce the method we adopt to defend against buffer overflows is named Proof-Carrying Code, or PCC for short, which is developed by Necula in U. C. Berkeley and Lee in Carnegie Mellon University. We present our effort to eliminate the buffer overflow problem with proper use of it in Section 4.

II. EXISTING PROTECTION SOLUTIONS

As the buffer overflow vulnerabilities are well-known, some solution proposals and implementations exist. In this section, we will pick up two effective ones and have a look at them, including Libsafe and PaX. In addition, the drawbacks of them will be pointed out at the end of this section.

A. Libsafe

A number of sensitive libc functions (strcpy, strcat, sprintf, vsprintf, getwd, gets, etc.) should always be inspected as the suspects of the buffer overflow vulnerabilities. The common bug of their implementations is that none of them performs bounds check while storing characters to the buffer. Thus the attackers may utilize them to break the system security. Libsafe [6], [7] is a library which rewrites all these sensitive functions to prevent any buffer overflow caused by a misuse of one of them. It launches alerts when a buffer overflow attempt is detected.

A typical implementation of strcpy is shown in Fig. 1:
The only criterion in the above implementation for deciding whether we should copy more characters into dest is the termination character ‘\0’. However, the buffer size of dest has not been taken into account as a factor and thus provides an opportunity for the attackers to inject code into dest and execute it.

The Libsafe version of these functions has been rewritten with bounds check performed and thus eliminates the misuse of them and potential buffer overflow attacks exploiting them. Generally we can’t tell the exact size of dest in strcpy since the caller has not provided it to us, while the distance between dest and the frame pointer can be used as the maximum size without compromising the integrity of the current activation record.

The preload mechanism of Linux is the basis for Libsafe to work. Libsafe intercepts the calls to the unsafe functions of standard C library and replaces them with its own implementation. While keeping the same semantics, the bound violation can be detected.

B. PaX

Under normal circumstances the stack and heap are purposed to hold data only. However, most cases of buffer overflow attacks need to inject code into the stack or heap and get it executed, which means that the stack and heap are used to hold code besides data. The basic idea of PaX [8] is to improve the paging mechanisms so that the stack and heap become non-executable and thus potential buffer overflow vulnerabilities can be detected. Tuning the stack and heap non-executable should be as easy as turning your hand over with the support of the segmentation mechanisms from i386 family processors. Unfortunately it becomes a non-trivial task under Linux since Linux relinquishes his rights to utilize the segmentation mechanisms offered by the processors.

To implement the notion of non-executable memory pages, PaX defines a system of states to represent the current condition of the pages. A state is defined by two permissions: execution and read/write, which indicates whether the page is used to hold code or data. Each permission may be granted to supervisor or user mode, or to none. This makes up nine possible states, among which six are considered as good states because they conform to the non-executability of a user mode page. The other three violating the rules are defined as bad states which should never occur [9]. Accordingly, each page has a current state, which may change as time goes by, depending on the memory page evolution. Similarly, PaX defines good transitions and bad transitions. A good transition is a transition to a good state, while a bad transition is a transition to a bad one, as the name suggested.

Armed with this knowledge, all we have to do is to prevent the memory pages from transiting to the bad states. To achieve the goal, it should come with no surprise that PaX inserts additional inspections according to its transition policy before passing the execution flow to the traditional page fault handler.

C. Drawbacks of the Existing Solutions

As we have pointed out earlier, although Libsafe and PaX are proved to be effective most of the time, both of them have their drawbacks. For example, imagine what if the program is linked statically with the standard C library? For this to be the case, Libsafe will be bypassed entirely. To escape the non-executable stack protection, a technique named returning into libc is invented. This is done by overwriting the return address of the exploited function with the address of a libc function (system(3) most of the time) rather than the buffer address where the shellcode resides. This can also lead to the execution of an unexpected function, without any attempt to execute code in the stack or in the heap [9]. What’s more? The performance decline is inevitable for the dynamic detection approaches such as PaX. Measurements show that an impact of around 5-8% slow down is perceivable [7].

III. PROOF-CARRYING CODE

The purpose of PCC is to provide a way for a host system to determine whether it is safe to install and execute a program derived from the code producer. This is accomplished by requiring that the code producer provides formal evidence, in some easy-to-verify form, that the program is well-behaved with respect to the safety policy published by the host [10].

Fig. 2 illustrates the overall architecture of PCC. Before the program from the code producer can get executed on the host system, it must go through four phases, which are certifying compilation, verification condition generation, proof generation and proof checking. On the right side of Fig. 2 we see the code producer’s part of the process. A certifying compiler is used to take a source program and compile it into annotated native code. The certifying compilation is largely conventional except for the output of some embedded annotations, which play an important role to instruct a one-pass inspection to generate the...
verification conditions. The verification conditions, or simply VC, have the property that their logical validity implies whether the code behaves safely. The proof generator then derives the verification conditions and constructs the logical representation of the proof in the logical system defined by the proof rules. The left side of Fig. 2 shows the host’s part of the process, which begins when a native-code binary and its proof are received from the code producer. In order to ensure that the given proof is exactly the safety proof for the given program, the host must also perform a one-pass inspection of the program to generate the verification conditions. Once the verification condition is obtained, the proof can then be checked to see whether it is indeed a valid proof of the verification conditions. Once the validation succeeds, the code is guaranteed to respect the safety policy and enabled to execute on the host with no more run-time checks [5], [10], [11], [12].

IV. PROBING BUFFER OVERFLOWS WITH PCC

Now let’s pick up a small example to run throughout the process to gain some intuition. The example, though small, is deliberately chosen in the sense that we need only a little change to make it represent the symptom of buffer overflows exactly. The sample C code and the corresponding annotated native code produced by the certifying compiler are shown in Fig. 3 and Fig. 4.

```c
int main()
{
    int i;
    char buf[12];
    for (i = 0; i < 12; ++i)
        buf[i] = 'A';
    return 0;
}
```

Figure 3. The sample C code

```assembly
main:
    pushl %ebp
    movl %esp,%ebp
    subl $16,%esp
    nop
    movl $0,-4(%ebp) ; i = 0
.L2:
    cmpl $12,-4(%ebp) ; i < 12?
    jl .L5 ; loop if yes
    jmp .L3
    .align 4
.L5: ANN_LOOP(
    INV = {(lt (sel4 rm (add ebp – 4)) 12),
    (ge (sel4 rm (add ebp – 4)) 0),
    MODREG = {edx, eflags, rm})
    leal -16(%ebp),%edx ; get &buf[0]
    addl -4(%ebp),%edx ; get &buf[i]
    movb $65,(%edx) ; buf[i] = 'A'
.L4:
    incl -4(%ebp) ; ++i
    jmp .L2
    .align 4
.L3: xorl %eax,%eax ; return 0
    jmp .L1
    .align 4
.L1: leave
    ret
```

Figure 4. The annotated native code

<table>
<thead>
<tr>
<th>NATIVE CODE</th>
<th>VC GENERATION</th>
<th>PROOF GENERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A2: (type rm mem)</td>
<td>(aidxi 1 (below1 (lt_b (geswap A5) A4)))</td>
</tr>
<tr>
<td></td>
<td>A3: (arrsize (add ebp -16) 12)</td>
<td>(lt (sel4 rm (add ebp – 16)) 12)</td>
</tr>
<tr>
<td></td>
<td>A4: (lt (sel4 rm (add ebp –4)) 12)</td>
<td>(ge (sel4 rm (add ebp – 4)) 0)</td>
</tr>
<tr>
<td></td>
<td>A5: (ge (sel4 rm (add ebp –4)) 0)</td>
<td>prove: (safewr1 (add (add ebp –16) (sel4 rm (add ebp –4))))</td>
</tr>
<tr>
<td>leal -16(%ebp),%edx</td>
<td></td>
<td>(wrArray1 A1 A2 szchar A3)</td>
</tr>
<tr>
<td>addl -4(%ebp),%edx</td>
<td></td>
<td>(aidxi 1 (below1 (lt_b (geswap A5) A4)))</td>
</tr>
<tr>
<td>movb $65,(%edx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L4:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>incl -4(%ebp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp .L2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5. The certification of the loop in the sample code
Figure 6. The proof rules used in Figure 5

Figure 7. The proof generation for safewr1

In the annotated native code, the ANN_LOOP annotation provides the loop invariant which declares a set of conditions that are claimed to hold every time the execution enters the loop. Now we will focus in detail on the loop in the sample code. The certification process for this segment is shown in Fig. 5. There are three columns in the figure, the first of which shows the annotated native code. The second column shows the VC generation along this control path and the third column shows the generated proof of the VC.

We have purposely thrown off the proof obligations irrelevant to our interest so that we can concentrate on the write operation to the buffer. As we see in Fig. 5, a proof obligation is generated to represent the sufficient safety requirement for the buffer write instruction: movb $65,(%edx). Then the proof obligation is proved under the current assumptions. The proof rules used in this example are shown in Fig. 6 and the proof generation process for safewr1 is detailed in Fig. 7. The basic idea is to ensure that the index is within the range of the buffer. It should come with no surprise that the proof obligation can be proved which indicates that this piece of code does behave safely.

Armed with the knowledge from the previous example, imagine what if the program shown in Fig. 3 is written as Fig. 8.

Notice that the only change we have made is the bigger loop boundary. Thus the annotation produced by the certifying compiler will also change accordingly. At this point what the ANN_LOOP annotation becomes is shown in Fig. 9. With the new assumptions we cannot prove that the index used to access the buffer in the loop is always within the range. The apparent result is that the buffer overflow vulnerability is discovered since the proof obligation cannot be proved.

The above example shows that PCC can be used to probe the obvious buffer overflow vulnerabilities appeared in the program, but what if the attackers choose to take advantage of the vulnerable functions from the standard C library? The most evident approach is brute forcing. We can recompile the library with the certifying compiler and then find out all the potential vulnerabilities which need to be rewritten the way Libsafe does. While this may be too costly, we had better look for another bullet. Fortunately PCC provides such a mechanism that the sensitive functions like strcpy can be protected by establishing preconditions for them. Upon calling such functions, the proper proof obligations are produced by the certifying compiler to ensure that certain arguments meet the special requirement of the functions. Taking strcpy as an example once again, the function prototype and the possible proof obligations upon such a function call is shown in Fig. 10.

```c
int main()
{
    int i;
    char buf[12];
    for (i = 0; i < 100; ++i)
        buf[i] = 'A';
    return 0;
}
```

Figure 8. A program with buffer overflow vulnerability

ANN_LOOP

INV = (((lt (sel4 rm (add ebp -4)) 100),
       (ge (sel4 rm (add ebp -4)) 0)),
       MODREG = [edx, eflags, rm])

Figure 9. The new ANN_LOOP annotation
It is the safety policy published by the host that determines the security rank of the host system. The safety policy comprises a set of proof-formation rules, along with a set of preconditions, which are constituted by the system administrator. Different host systems may have different safety policies to reflect the special security requirement on the given system. Thus the PCC method provides us a way to keep track of the coming-up variants of the buffer overflow vulnerabilities. Armed with this adaptability, we are well prepared to handle the new variants of the buffer overflow problems since the only thing we need to do at this point is to enlarge the existing safety policy with our countermeasure.

The careful reader has probably noticed that the PCC method is different from most of the existing solutions in the way that it takes a static analysis approach. Once the code gets across the proof checking, it is qualified to run on the host system with no more run-time overhead. Actually, the certification process is very similar to the way that it takes a special approach. In particular each loop body is executed symbolically once, which means that even a non-terminating program can be executed symbolically in a finite amount of time.

V. CONCLUSION

The inspiration for introducing PCC into the realm of defending against buffer overflow attacks comes from the discovery that most of the existing solutions take a dynamic approach, which is destined to introduce run-time overhead. In addition, we find through a brief retrospection of some effective ones, such as Libsafe and PaX, that the existing solutions to this problem have focused on the execution environment of the malicious program rather than the hypostasis of buffer overflow. Thus most of them can be bypassed. Actually, the rationale for buffer overflow is very simple and it is quite efficient to detect such vulnerabilities through a static analysis approach. This is where PCC finds its place. We have shown that the buffer overflow programs cannot go through the certification process since the proof obligations cannot be proved, no matter whether the program contains the apparent buffer overflow code or takes advantage of the vulnerable libc functions. Apart from this, the PCC method is well prepared to handle the coming-up variants of the buffer overflow problems because we can adapt the safety policy to reflect our countermeasure to the new problems.

ACKNOWLEDGMENT

This work was supported by National Natural Science Foundation of China (No. 61272167), National High Technology Research and Development Program of China (863) (No. 2011AA01A204) and National Science and Technology Infrastructure Program (No. 2012ZX01039-004).

REFERENCES