A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention

John Wilander & Mariam Kamkar

Presented by: PU LI

Introduction

• The size and complexity of software system is growing, increasing the bugs
• More vulnerabilities, since the same mistakes are made over and over again
• Buffer overflow still stands for about 50% of the vulnerabilities reported by CERT
• One starting point: developing tools that can be applied directly to the source code and solve or warn about security vulnerabilities.

Software Vulnerabilities reported to CERT (1995-2001)

Purpose of the Paper

• Investigate the effectiveness of four publicly available tools for dynamic prevention of buffer overflow attacks
  - StackGuard
  - Stack Shield
  - ProPolice
  - security library Libsafe/Libverify

Attack Methods

• An attack aims to change the flow of control, letting the attacker execute arbitrary code.
• It involves two steps for an attacker:
  - Injecting attack code or attack parameters into some memory structure of the vulnerable process
  - Abuse some vulnerable function to alter data that controls execution flow
• How to change the flow of control?
  - change the code pointer

Memory Layout in UNIX

• Memory layout of a UNIX process
  - High address
  - Stack
  - Heap
  - BSS segment
  - Data segment
  - Text segment
• The UNIX stack frame
  - Lower address
  - Local variables
  - Old base pointer
  - Return address
  - Higher address
  - Argument
**Attack Targets**

- The return address, *allocated on the stack*
- The old base pointer, *allocated on the stack*
- Function Pointer, *allocated on the heap, in the BSS or data segment, or on the stack either as a local variable or as a parameter*
- Longjmp Buffer

**Buffer Overflow Attacks**

- Extra data “spill over” into the adjacent memory structure, overwriting anything that stored there before
- How to change the flow of control
  - redirect the return address to attack code
  - build a fake stack frame with return address pointing to attack code, then overwrite the old base pointer
  - function pointer is redirected to attack code
  - `setjmp()` & `longjmp()` are called

**Intrusion Prevention**

- Static
  - find security bugs in the source code so that the programmer can remove them
- Dynamic
  - change the run-time environment or system functionality making vulnerable programs harmless, or at least less vulnerable

**StackGuard**

- **Goal**
  - Designed for detecting and stopping stack-based buffer overflows targeting the return address
- **Key idea**
  - buffer overflow attacks overwrite everything on their way towards their targets
- **How to implement?**
  - Place a dummy value in between the return address & the stack address above

**Discussion**

- What is the main problem or limitation of StackGuard?
  - Only stop overflow attacks that overwrite everything along the attack
  - Attacker can still abuse a pointer…pointing to return address
  - So…XOR of the canary and return address
Stack Shield

- Three versions:
  -- Two against overwriting of the return address
    Global Ret Stack
    Ret Range Check
  -- One against overwriting of function pointers

Global Ret Stack

- a separate stack for storing the return address of functions called during execution
- Discussion: What is the limitation or problem of it?
  - only prevention and no detection
  - security of the separate stack

Ret Range Check

- Simpler and faster
- a global variable to store the return address of current function
- before returning, return address is compared with the stored copy in the variable
- Note:
  - It can detect!

Third version

- no function pointer is allowed to point into other parts of memory than the text segment
- It is impossible for an attacker to make it point at code injected to the process
- Why?
  Injection of data only can be done into the data segment, the BSS segment, the heap or the stack

ProPolice

- Re-allocate the variables by rearranging the local variables so that char buffers always are allocated at the bottom

<table>
<thead>
<tr>
<th>Lower address</th>
<th>Higher address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local variables and pointers</td>
<td>Return address</td>
</tr>
<tr>
<td>Local char buffers</td>
<td>Old base pointer</td>
</tr>
<tr>
<td>Guard Value</td>
<td>Argument</td>
</tr>
</tbody>
</table>

Libsafe and Libverify

- Libsafe
  - Provides a combination of static and dynamic intrusion prevention
  - Range check is made before function call
- Libverify
  - Uses a similar dynamic approach to StackGuard
Libsafe

- Estimate a safe boundary for buffers, and check this boundary before any vulnerable function is allowed to write to buffer.
- No local variable is allowed to expand further down than the beginning of the old base pointer.

<table>
<thead>
<tr>
<th>Lower address</th>
<th>Boundary address</th>
<th>Higher address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local variables</td>
<td>Old base pointer</td>
<td>Return address</td>
</tr>
<tr>
<td>Argument</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Libverify

- Key idea:
  - Alter all functions in a process.
  - First thing done in function: copy the return address onto a canary stack located on the heap.
  - Last thing done before returning: verify the return address by comparison.
- Discussion: What is the potential problem?
  - It doesn’t protect the integrity of canary stack.

Testbed

- A testbed of twenty buffer overflow attack forms.
- Define an attack form as a combination of a technique, a location, and an attack target.
  - Techniques:
    - Either we overflow the buffer all the way to the attack target or we overflow the buffer to redirect a pointer to the target.
  - Locations:
    - stack, or the heap/BSS/data segment.
  - Attack Targets:
    - The return address, the old base pointer, function pointer, and longjump buffers.

Empirical test

<table>
<thead>
<tr>
<th>Development Tool</th>
<th>Attacks prevented</th>
<th>Attacks halted</th>
<th>Attacks missed</th>
<th>Abnormal behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>StackGuard Terminator Canary</td>
<td>8 (8%)</td>
<td>4 (30%)</td>
<td>16 (80%)</td>
<td>1 (5%)</td>
</tr>
<tr>
<td>StackGuard Random XOR Canary</td>
<td>8 (8%)</td>
<td>6 (30%)</td>
<td>11 (70%)</td>
<td>1 (5%)</td>
</tr>
<tr>
<td>Stack Shield Global Ret Stack</td>
<td>6 (30%)</td>
<td>7 (35%)</td>
<td>9 (35%)</td>
<td>1 (5%)</td>
</tr>
<tr>
<td>Stack Shield Global &amp; Range Check</td>
<td>6 (30%)</td>
<td>7 (35%)</td>
<td>9 (35%)</td>
<td>1 (5%)</td>
</tr>
<tr>
<td>ProPolice</td>
<td>8 (10%)</td>
<td>3 (15%)</td>
<td>9 (85%)</td>
<td>1 (5%)</td>
</tr>
<tr>
<td>Libsafe and Libverify</td>
<td>8 (10%)</td>
<td>6 (30%)</td>
<td>14 (60%)</td>
<td>1 (5%)</td>
</tr>
</tbody>
</table>

Theoretical Comparison

- Interesting results?
  - The most efficient, here, ProPolice, can only prevent 50% of attack forms.
  - Only ProPolice & Stack Shield offer real runtime prevention…others are more or less detection systems…

Common Shortcomings

- Four generic problems for considering future research:
  - Denial of Service
  - Storage Protection
    (Canaries or separate return address stacks have to be protected also)
  - Recompilation of Code
  - Limited Nesting Depth
    (When keeping a separate stack with copies of return addresses, the nesting depth of the process is limited)
Conclusion

- In practice, at best 40% of the attack forms are prevented, another 10% detected and halted, leaving 50% of the attacks still at large
- Combining techniques in theory, nearly a third of attacks are still missed
- Due to the general weakness of Dynamic Intrusion Prevention solution – tools aim to protecting known attacks, not all targets…

Questions & Comments……