To Detect Stack Buffer Overflow With Polymorphic Canaries

何钟灵

April 29, 2018

1 Personal

1.1 intro

This is based on an essay by Zhilong Wang in our group. Our group is named SECLAB in Lab 428, Building of Computer Science and Technology, Nanjing University.

2 Background

2.1 Introduction

Buffer overflow exploit is an anomaly where a program, while writing data to a buffer, overruns the buffer’s boundary and overwrites adjacent memory locations.

Example:

<table>
<thead>
<tr>
<th>variable name</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>[null string]</td>
<td>1979</td>
</tr>
<tr>
<td>hex value</td>
<td>00 00 00 00 00 00 00 07 BB</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>variable name</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>'e' 'x' 'c' 'e' 's' 's' 'i' 'v'</td>
<td>25856</td>
</tr>
<tr>
<td>hex</td>
<td>65 78 63 65 73 73 69 76 65 00</td>
<td></td>
</tr>
</tbody>
</table>
Stack-based exploitation If that happened to a stack, the result would be that some variables in the stack Example:

<table>
<thead>
<tr>
<th>variable name</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td></td>
<td>[null string]</td>
</tr>
<tr>
<td>hex value</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

Stack Smashing Exploitation If that happens to the stack, attacker can modify saved rip and return to his exploit payload.

3 Previous approach

3.1 Canary

Stackguard: Canary Store a canary value below return address. Before returning, check if it was modified.
Stackguard GCC has different compiling options to ensure different levels of security:

- `-fstack-protector` protects only some vulnerable functions.
- `-fstack-protector-strong` strikes a better balance between security and performance.
- `-fstack-protector-all` protects all functions whether they need it or not.

Another possible solution: Save return address.

- TLS has limited space, therefore, return address cannot be saved in TLS.
- Shadow Stack saves address in another shadow stack to solve this problem.

Two different shadow stacks. Traditional vs parallel shadow stacks.

Shadow Stack and Canary Shadow Stack has a higher overhead, but is more secure.
Stackguard: Byte-by-byte Attack Since each thread has the same canary value, we could try each byte. If the program doesn’t crash, it means the value is correct.

64bit canary: 64bit = 8 * 8bit
Worst case: 8 * 2^8 = 2048 trails
Average: 8 * 2^8/2 = 1024 trails

DynaGuard: Renew after fork Upon fork() is finished, renew the stack canary in stack and TLS. Therefore, canary of each child process / thread would be different. But it would fail when the child process returns to the frames inherited by the parent process.

Therefore, solving this problem increases its overhead.
4 Polymorphic Canary

4.1 canary

Polymorphic Canary It would be nice if we can redesign the stack guard which has the same security level as DynaGuard and same overhead as SSP Canary. Polymorphic Canary or Polymorphic Stack Smashing Protection (P-SSP) would achieve this.

PSSP: Design let \( Canary_{TLS} = C0 \oplus C1 \)

\( C0 \) is a random 32-bit integer

\( C1 = Canary_{TLS} \oplus C0 \)

This would ensure that \( C0 \) and \( C1 \) would never be the same. And they would not leak information of Canary.

PSSP: Security Wang has proved that byte-by-byte attack is impossible for PSSP.

For compatibility, PSSP use 32 bit canary in 64 bit system while using Binary Instrumentation. Hence, attacker need \( 2^{32} \) trails to successfully guess TLS canary.

PSSP: Overhead

<table>
<thead>
<tr>
<th>prologue of PSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

epilogue of SSP vs epilogue of PSSP
obviously, 2 instructions doesn’t cause much overhead

PSSP: Implementation

- PSSP shared library
  exports setup_p-ssp, fork, and pthread_create to override their counterparts in the standard GNU C library.

- Source code level: Compiler plugin
  LLVM FunctionPass is invoked on each function in the source code, perform instrumentation on the intermediate language (IR).

- Binary level: Binary Instrumentation for P-SSP
  In order to preserve stack layout, there will only be 64bit left for all the canaries. Wang decided to downgrade from 64bit canary to 32bit canary. 
  The space for function epilogue is limited, as a result, Wang use call to reduce code size.

PSSP: Binary Instrumentation

PSSP: Evaluation
PSSP has a maximum of 2.5% overhead

PSSP: Evaluation

Conclusion: PSSP has a lower overhead comparing to Shadow Stack.