How to Make ASLR Win the Clone Wars: Runtime Re-Randomization

Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee

Georgia Tech, CISPA, Saarland University, MPI-SWS, DFKI
What did we do?

- We re-randomize the memory layout of the cloned (*i.e.*, forked) processes at **runtime**
In this talk, I will explain...

• **Why** we need to re-randomize cloned processes?
  – To prevent clone-probing attacks

• **How** to re-randomize them?
  – A semantic-preserving and runtime-based approach

• **What** are the results?
  – Defeated clone-probing, e.g., Blind ROP attack
  – No performance overhead to cloned processes
Background - ASLR

- Address Space Layout Randomization (ASLR)
  - Mitigating code reuses attacks, privilege escalation, and information leaks

![Diagram of ASLR runtime behavior across three runs: Run 1 shows code and data, Run 2 shows data and code, Run 3 shows code and data. Dashed arrows indicate variation in memory layout between runs.](image)
Background - ASLR

- Address Space Layout Randomization (ASLR)
  - Mitigating code reuses attacks, privilege escalation, and information leaks

- One time, per-process, load-time
Background – Daemon Servers

- Web services are powered by daemon servers, e.g., Nginx web server
1) The daemon process pre-forks multiple worker processes that handle users requests
Designs of Daemon Server

1) The daemon process pre-forks multiple worker processes that handle users requests.
2) The daemon will re-fork a new worker process if it crashes, to be robust.
Designs of Daemon Server

2) The daemon will re-fork a new worker process if it crashes, to be robust.

All forked worker processes share the same memory layout as the daemon process.
When ASLR meets daemon servers...
Clone-Probing Attack

- Attack goal: guess the randomized address (e.g., return address), say a web server with a stack buffer overflow vulnerability.
Clone-Probing Attack

- Attack goal: guess the randomized address (e.g., return address), say a web server with a stack buffer overflow vulnerability.

Stack in remote server:

- Buffer:
  - AAAA
  - AAA

- Return address:
  - 12 34 56 78 9a bc ed f0

- Stack in remote server:
  - 00 34 56 78 9a bc ed f0

Attack payload:
- AAAA

Crash, try another one.
Clone-Probing Attack

- Attack goal: guess the randomized address (e.g., return address), say a web server with a stack buffer overflow vulnerability
Clone-Probing Attack

- Attack goal: guess the randomized address (e.g., return address), say a web server with a stack buffer overflow vulnerability

```
<table>
<thead>
<tr>
<th>Buffer</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAAAAAA</td>
<td>12 34 56 78 9a bc ed f0</td>
</tr>
<tr>
<td>AAAAAAAA</td>
<td>00 34 56 78 9a bc ed f0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>AAAAAAAA</td>
<td>01 34 56 78 9a bc ed f0</td>
</tr>
</tbody>
</table>
```

Crash, try another one
Crash, try another one
Bingo, continue to guess next byte
Clone-Probing Attack

• Attack goal: guess the randomized address (e.g., return address), say a web server with a stack buffer overflow vulnerability

Stack in remote server

Attack payload

```
AAAAAAA
AAAAAAA
...
AAAAAAA
AAAAAAA
Return address
12 34 56 78 9a bc ed f0
```

```
00 34 56 78 9a bc ed f0
01 34 56 78 9a bc ed f0
... ...
12 34 56 78 9a bc ed f0
12 00 56 78 9a bc ed f0
... ...
```

Crash, try another one
Crash, try another one
Bingo, continue to guess next byte
Clone-Probing Attack

- Attack goal: guess the randomized address (e.g., return address), say a web server with a stack buffer overflow vulnerability

![Stack in remote server diagram with attack payload and return address]

- Attack payload
  - Stack in remote server
  - Return address
  - 12 34 56 78 9a bc ed f0
  - Crash, try another one
  - Crash, try another one
  - Bingo, continue to guess next byte
  - Finally, get all bytes
Clone-Probing Attack

- Attack goal: guess the randomized address (e.g., return address) to crash the remote server with a stack buffer overflow vulnerability...

...buffer...

Stack in remote server

...Attack payload

Crash, try another one

Crash, try another one

Bingo, continue to guess next byte...

...AAAAAAA

AAAAAAA 00 34 56 78 9a bc ed f0

AAAAAAA 12 34 56 78 9a bc ed f0

AAAAAAA 12 00 56 78 9a bc ed f0

Finally, get all bytes

Brute-forcing complexity is reduced from $2^{64}$ to $8 \times 2^8$ (From thousands of years to 2 minutes 😊)
This Attack is Critical!

A simple buffer overflow ➔ bypass ASLR (two minutes) ➔ control daemon server 😞
Preventing clone-probing with RuntimeASLR

Solution: re-randomizing the memory layout of cloned processes
Challenge

• Remapping memory → dangling pointers

• How to track all pointers on the fly and update them?
  – Accuracy
  – Efficiency
Pointer Tracking Problem

• Treat it as a taint tracking problem
Source Pointers

- Kernel routinely loads program
  - Easy to find source pointers
- Only in stack and registers
Pointer Tracking Policy

Source pointers + Pointer tracking policy = All tracked pointers
Pointer Tracking Policy

- Read 1,513 pages Intel ISA manual and manually define them??
Automatic Tracking Policy Generation

• Automatically identifying instructions behaviors

• This way, we know if it generates or destroys some “values”
How to Determine a Pointer?

• Without type info, how do we know if a value is a pointer?

• Example: `mov rdi, rsp`
  – Before: `rsp=0xcafebabe`, and know it is a pointer
  – After: `rdi=0xcafebabe`, memory is unchanged
  – How to know if `rdi` is a pointer?
Multi-Run Pointer Verification

- Observation: \textit{rdi} is \textbf{likely} a pointer if it points to mapped memory on 64-bits platform, why?
- Run program \textit{n} times with \textbf{ASLR}, if \textit{rdi} always points to mapped memory, \textit{rdi} is \textbf{more and more likely} a pointer
  - Mapping \textit{n} runs with instruction execution sequence

![Diagram of multi-runs with ASLR-enabled]
Accuracy of Multi-Run Verification

- Assume size of mapped memory is $b$ bytes, run $n$ times on 64-bits platform, false positive rate for one value is:

$$P_{fpr} = \left( \frac{b}{2^{64}} \right)^n = b \cdot 2^{-64} \cdot n$$
Accuracy of Multi-Run Verification

• Assume size of mapped memory is \( b \) bytes, run \( n \) times on 64-bits platform, false positive rate for one value is:

\[
P_{fpr} = \left( \frac{b}{2^{64}} \right)^n = b \cdot 2^{-64} \cdot n
\]

• Say \( b \) is 22MB (Nginx) and run 2 times. This will result in FPR=\( 2^{-103} \)
Export Policy

• Given `mov reg1, reg2`
  – if `reg2` is a 64-bits register and tainted (i.e., a pointer) ➔ taint `reg1` after execution
Track All Pointers

Source pointers + Pointer tracking policy = All tracked pointers
Implementation

• Intel’s PIN—a dynamic instrumentation tool

• Three modules

  - Policy generator (pintool)
  - Pointer tracker (pintool)
  - Randomizer (shared lib)

• Source code
  - Coming soon
Evaluation

• Correctness
  – Applied to Nginx web server
  – Memory snapshot analysis to find all pointers
  – RuntimeASLR correctly finds all pointers
Evaluation

• Security
  – Blind ROP is a clone-probing attack
  – Addresses of all modules are re-randomized
  – RuntimeASLR successfully defeats it
Evaluation

• Performance
  – Pointer tracking is extremely expensive: >10,000 times on SPEC CPU2006
    • One time overhead at startup; 35 seconds for Nginx
  – However, no overhead on cloned worker processes

![Graph showing response times for different numbers of concurrent clients.](image-url)
Discussions and Limitations

- Ambiguous policy
- Completeness of tracking policy
- Applicability for general programs
- Supporting pointer obfuscation
Demo

• Defeat Blind ROP attack with RuntimeASLR
Recap

• Clone-probing attacks $\rightarrow$ bypass ASLR $\rightarrow$ control daemon server or steal sensitive data

• We proposed RuntimeASLR to defeat clone-probing attacks
  – Automatic pointer tracking policy generation
  – Support COTS binaries, no system modifications
  – No overhead to cloned worker processes (after fork())
Recap

• Clone-probing attacks $\rightarrow$ bypass ASLR $\rightarrow$ control daemon server or steal sensitive data

• We proposed RuntimeASLR to defeat clone-probing attacks
  – Automatic pointer tracking policy generation
  – Support COTS binaries, no system modifications
  – No overhead to cloned worker processes (after fork())

THANKS
Backup slides
Pointer Tracking Approaches

• Compiler-based instrumentation
  – Pros: type info, efficient in tracking
  – Cons: type-confusion, hard to decouple instrumentation, require source

• Dynamic instrumentation
  – Pros: easy to decouple instrumentation, support COTS
  – Cons: lack of type info, tracking is expensive
Pointer Tracking Approaches

• Compiler-based instrumentation
  – Pros: type info, efficient in tracking
  – Cons: type-confusion, hard to decouple instrumentation, require source

• Dynamic instrumentation
  – Pros: easy to decouple instrumentation, support COTS
  – Cons: lack of type info, tracking is expensive
Accuracy of Multi-Run Verification

• Assume $b$ instructions in $b$ bytes memory. Probability for at least one non-pointer value misidentified as a pointer is:

$$1 - (1 - P_{fpr})^b \iff b^2 \cdot 2^{-64} \cdot n$$
Accuracy of Multi-Run Verification

• Assume $b$ instructions in $b$ bytes memory. Probability for at least one non-pointer value misidentified as a pointer is:

$$1 - (1 - P_{fpr})^b \iff b^2 \cdot 2^{-64} \cdot n$$

• Say $b$ is 100MB and run 2 times. This will result in FPR=$2^{-76}$