Isomeron

Code Reuse Attacks and Defenses against them

Sebastian Kulesz
Some background

- ROP
  - ASLR
  - Fine-grained ASLR
- JIT-ROP
  - Oxymoron
ROP

Requires static analysis of the binary.

1. Hijack application flow
2. Execute pre-built gadget chain
3. ...
4. Profit?
Mitigating ROP

- **ASLR**
  - Randomize the base addresses of code and data segments
  - Can be bypassed with memory disclosure vulnerabilities

- **Fine-grained ASLR**
  - ASLR with different granularity
  - Function, BBL, or instruction location
  - Randomizes internal structure of an application
JIT-ROP

Works around the static analysis needed for ROP. Generates the chain on-the-fly using the scripting environment of the application.

1. Exploit information disclosure vulnerability
2. Disassemble memory pages
3. Generate gadget chain on-the-fly
4. Execute gadget chain
5. ...
6. Profit?
JIT-ROP
JIT-ROP

Uses a *single* leaked runtime address to disassemble the content of hundreds of memory pages recursively and generates ROP exploits on-the-fly.

- Bypasses ASLR
- Bypasses fine-grained ASLR

Addresses are obtained at runtime, so they remain valid during the execution of the chain.
Mitigating JIT-ROP

**Oxymoron**: Hides code references encoded in direct branch instructions.

- Takes advantage of x86 segmentation to prevent the disclosure of memory pages.
- Adds a level of indirection by storing addresses in a table.
- Separation between *user-land* and *kernel-land* prevents an adversary from resolving the address.
  - A segment register is used as an index.
- Broken by using **indirect memory disclosure**
Breaking Oxymoron

Real-world exploit:

- Internet Explorer 8:
  - Assume it is protected with an Oxymoron mechanism
  - Well-documented heap-based buffer overflow vulnerability

- We require:
  - Memory error
  - Information disclosure
Breaking Oxymoron

1. Heap feng shui:
   ○ Carefully arranging objects close to each other

2. Overflow the vulnerable buffer
   ○ Overwrite the string length

3. We gained access to the vtable pointer
Breaking Oxymoron

1. Recursively discover and disassemble code pages
2. Search for gadgets
   - If possible, look for a stack pivot gadget
   - Look for a syscall gadget
3. Mark a controlled page as executable to execute shellcode
4. Hijack control flow by tampering with the vtable
5. ...
6. Profit?
Breaking Oxymoron

- It only protects code pointers in branch instructions
- Memory information disclosure can come from somewhere else
Isomeron

Code randomization resilient to JIT-ROP

A. Design Decisions
B. Assumptions
C. Adversary Model
D. Security Objectives
E. High-Level Idea
F. Architecture

Implementation
A. Design Decisions

Evaluation of related approaches:

- **Constant re-randomization:**
  - Adversary can exploit time frame between subsequent randomizations

- **Instruction-Set Randomization:**
  - Encrypts code using a secret key
  - Prevents the disassembly of code at runtime
  - Uses XOR which is vulnerable to known-plaintext attacks
  - Using AES introduces an unacceptable performance degradation
B. Assumptions

- **Non-Executable Memory**:
  - \( W \oplus X \) enabled by default in modern OSs

- **Fine-Grained ASLR**:
  - Gadgets contained in the original code image either:
    - Reside at a different offset
    - Eliminated by replacing instructions with an equivalent instruction
    - Broken due to instruction reordering or register replacement

- **Trust in the Diversifier**:
  - Adversary can’t tamper with Isomeron
  - Availability of a trusted pool of randomness
C. Adversary Model

● **Exploiting memory vulnerabilities:**
  ○ The adversary has knowledge of a vulnerability

● **Full memory disclosure:**
  ○ Access to all code pages of an application
  ○ Can circumvent fine-grained ASLR

● **Bruteforcing:**
  ○ Limited number of attempts for the attack
D. Security Objectives

Mitigate traditional JIT-ROP attacks

- **Protection against traditional ROP:**
  - Construction of the gadget chain prior to the execution of the program
  - Mechanism that changes the address of gadgets and breaks gadget chains.

- **Protection against JIT-ROP:**
  - Mechanism to counter memory disclosure vulnerabilities.
D. Security Objectives (2)

- Protection against ret-to-libc and JOP:
  - Drastically reduce the success probability of ret-to-libc
  - Same techniques apply to JOP

- Protection of diversifier:
  - Protect Isomeron from being compromised
E. High Level Idea

JIT code reuse relies on:

- Disclosing memory to construct a payload
- Exploit a vulnerability to execute generated payload

We need to ensure that the addresses of the gadgets in the chain change after it is built.

This will result in undefined behaviour, and likely lead to a crash.
E. High Level Idea

Simultaneously load 2 copies of the program in the same address space:

- One copy is the original code
- The other copy is diversified using any fine-grained ASLR technique

At runtime, we flip a coin to decide which copy of the program should be executed next.

Gadget sets are completely different in each copy of the program, and an attacker can not reliably predict which one will be executed. Thus preventing the adversary to construct a payload, even with full knowledge of the memory contents.
E. High Level Idea
F. Architecture

- **Program Twinning**: To clone a program within the same virtual address space, they applied dynamic binary instrumentation on the granularity of basic blocks.

- **Twin Diversification**: Apply fine-grained ASLR. A ROP sequence should never reside at the same offset in the original and diversified version.

- **Coin-Flip Instrumentation**: Execution randomization at the granularity of function calls. Only way to preserve original semantics.
F. Architecture

Instrumentation for function-calls and function-returns:
F. Architecture

- **Instrumentation of direct function calls:**
  - Ensure that we take control of the execution flow at a function call
  - Overwrite function call to execution diversifier
  - Flip a coin and decide which version will be executed

- The execution diversifier calculates the offset to be added:
  - 0 if execution should continue in the original image
  - Distance between program images otherwise

- The return address should always point to the original image

- Record each random decision to return to the correct image
F. Architecture

- **Instrumentation of function returns:**
  - The diversifier takes control whenever the program issues a return instruction.
  - The return address is read from the stack and used to determine the correct origin, and adjusted if necessary.
  - Only the diversifier knows if the return address was adjusted. It is unknown to an adversary.
F. Architecture

- **Instrumentation of indirect jumps and calls:**
  - Handled similarly to direct branches
  - The destination address is calculated at runtime
Implementation of Isomeron

Using **Dynamic Binary Instrumentation:**

- All control transferring instructions are modified such that the DBI software controls which instruction is executed next.
- Works on the granularity of a Basic Block

In contrast to static instrumentation, DBI has access to runtime information.

At the end of a translated basic block an exit stub is emitted. The exit stub saves the current execution context and transfers the control to the instrumentation framework, which contains the runtime information needed to calculate the address of the next basic block.
Implementation of Isomeron
Implementation of the DBI framework

1. Fetch basic block from the original binary
2. Analyze, translate, and emit them into a code cache
3. Emit a diversified block into a separate code cache

The execution diversifier switches between both code caches by flipping a coin.
Implementation of the DBI framework

- **Setup**: Implemented as a DLL
  - Injected into the desired process
  - Target application started in suspended mode. Execution stops right before the ‘main’ function
  - Opportunity for the library to take over and perform the instrumentation

- **Initialization of code and data**:  
  - Needs to have thread support without loss of performance
  - Access the thread’s private memory with hardcoded addresses
  - Copy each function which accesses the private memory, and adjust the references to said region
Implementation of the DBI framework

Basic block translation:

- Loader stops execution right before the first basic block of the application.
- Disassemble and analyze it’s code
- Modifications applied during translation:
  - Control flow instructions
    - Exit stub emission
  - Unaligned gadgets:
    - Prevent the adversary from executing unaligned code
  - Path and code diversification
    - Redirect call and return to execution diversifier
Implementation of the DBI framework

Execution diversifier:

- To preserve semantics, jumps are never randomized
- The pool of randomness is pre-computed for efficiency
- Can be re-randomized to prevent side-channel attacks
- Coin-flip results are saved in a per-thread private memory
Security Considerations

- Security based on uncertainty to predict the outcome of a random decision
- Adversary can’t predict which gadget chain will be executed after the control flow is hijacked
Security Considerations

Mitigating (JIT-)ROP:

- Assume that adversary can disclose the address space and assemble the gadget chain using the original and diversified copy of the application.
- Before diverting control flow, the chain needs to contain all the addresses of the gadgets that will be executed.
- The chain is unmodifiable after the control flow is diverted.
- Each image will contain different gadgets.
- The adversary has a probability $p=0.5$ of guessing the right image for each gadget.
- Probability decreases exponentially with chain length.
Security Considerations

Preventing ret-to-libc:
- Can not defeat it completely
- We only limit the number of possible ret-to-libc targets

Mitigating JOP:
- Limit potential jump target addresses
- Only allow jumps to benign target addresses

Disclosure of diversifier data: Prevented by using segmentation

Return to unaligned instructions: Can only return to an instrumented BBL
Conclusions

- Effective technique for mitigating most ROP-based attacks.
- Incurs in a performance overhead of ~19%
- Renders CPU-based optimizations useless
- Duplicates code memory usage of any instrumented application
References

1. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming
2. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization
3. M. Backes and S. Nurnberger. Oxymoron - making fine-grained memory randomization practical by allowing code sharing.
Discussion

- Possible optimization (ie: Novell branch prediction)
- Mitigation against JIT Spraying
- Possible enhancements