ASLR-Guard:

Stopping Address Space Leakage for Code Reuse Attacks

Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung,
Taesoo Kim, Wenke Lee

School of Computer Science
Georgia Tech

Code Reuse Attack

- Circumvent DEP or W^X
  - Code reuse is usually the only way to launch "remote code execution" attacks
  - It is prevalent in real world

Code Reuse Attack

- Circumvent DEP or W^X
  - Code reuse is usually the only way to launch "remote execution" attacks
  - It is prevalent in real world

Browsers

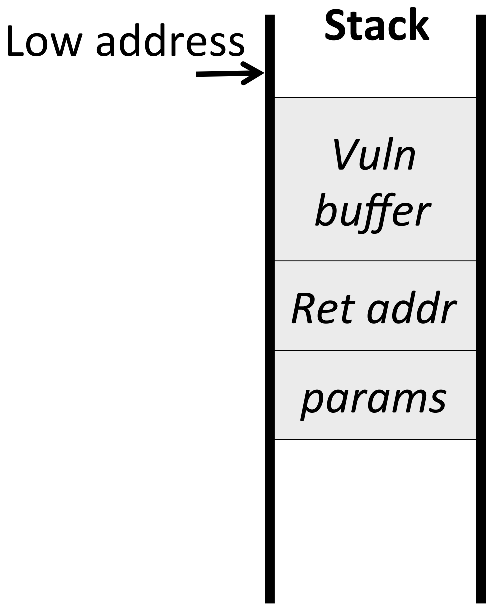
Apache HTTP SERVER

Servers

Kernels

Attackers

A Code Reuse Example



Stack

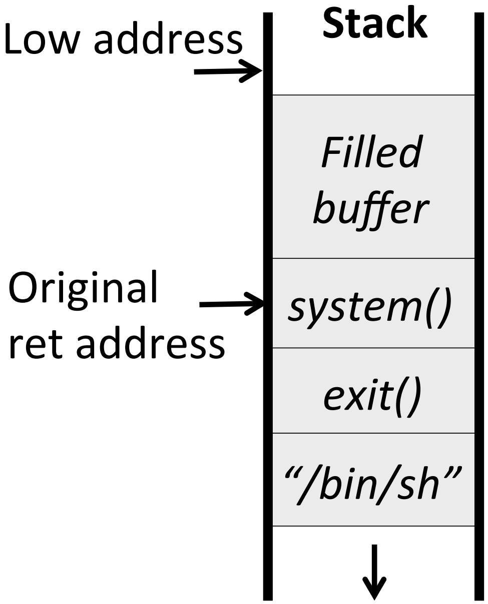
Low address

Vuln buffer

Ret addr

params

A Code Reuse Example



Low address

Stack

Filled buffer

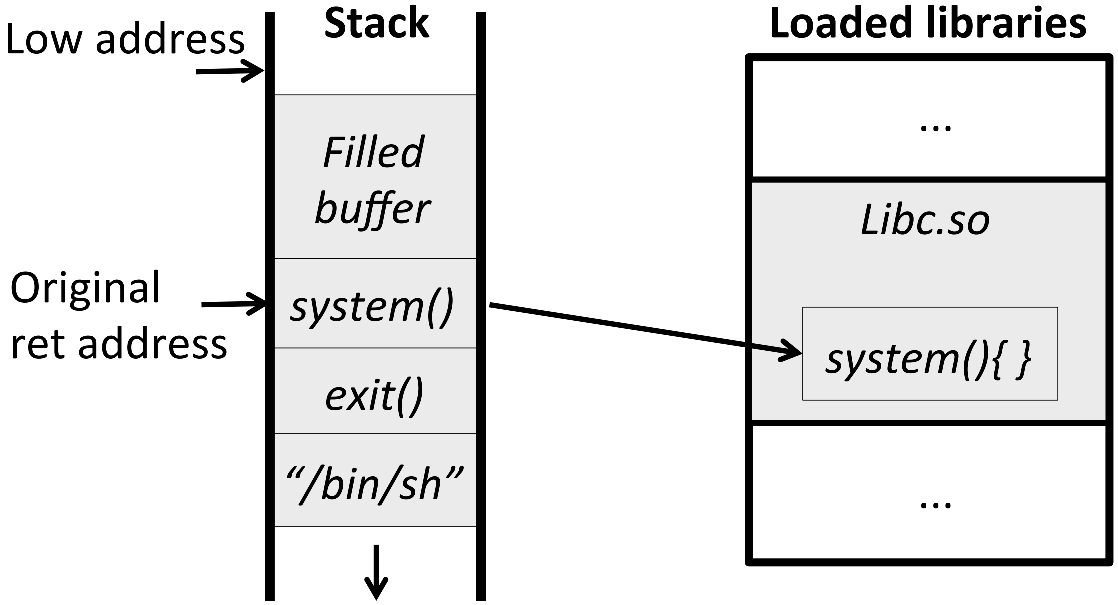
Original ret address

system()

exit()

“/bin/sh”

A Code Reuse Example



Code Reuse Attacks Becoming More Sophisticated

- More flexible, more automated, and more difficult to detect and defend against

<2001
Return-into-libc
》
2007
Return-oriented Programming
》
2010
JOP/ROP without returns
》

》
2013
JIT-ROP
》
2014
Signal ROP
PHP ROP
》
2015
COOP
Control Jujutsu
》
… ?

It’s Easy to Launch Code Reuse Attacks

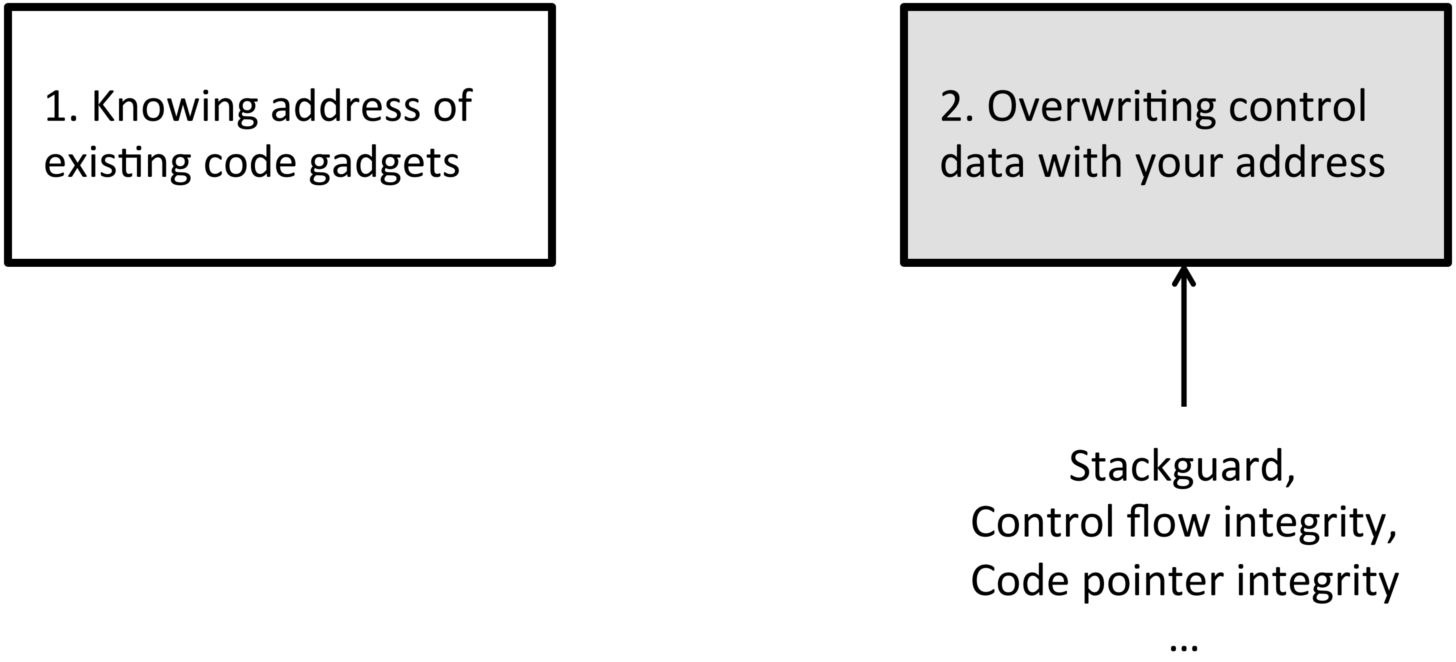
- Two typical requirements

1. Knowing address of existing code gadgets

2. Overwriting control data with your address

It’s Easy to Launch Code Reuse Attacks

- Two typical requirements



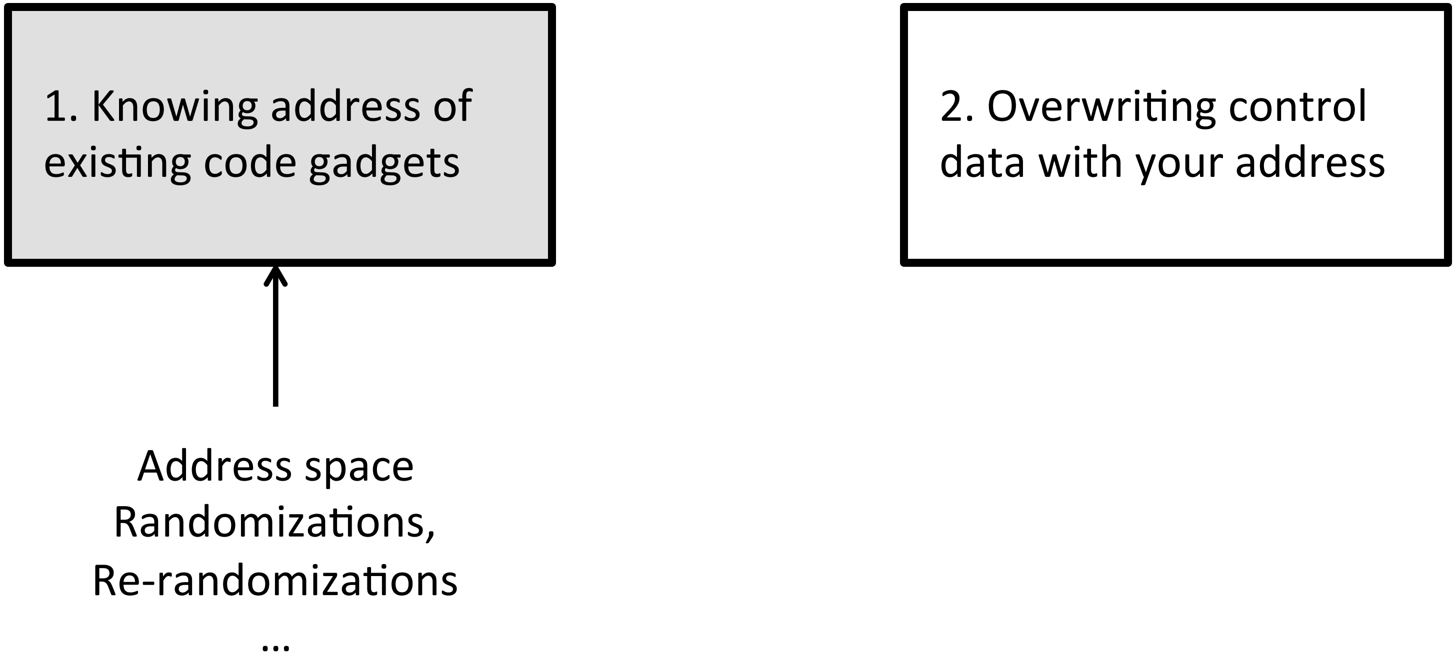
1. Knowing address of existing code gadgets

2. Overwriting control data with your address

Stackguard,
Control flow integrity,
Code pointer integrity
…

It’s Easy to Launch Code Reuse Attacks

- Two typical requirements



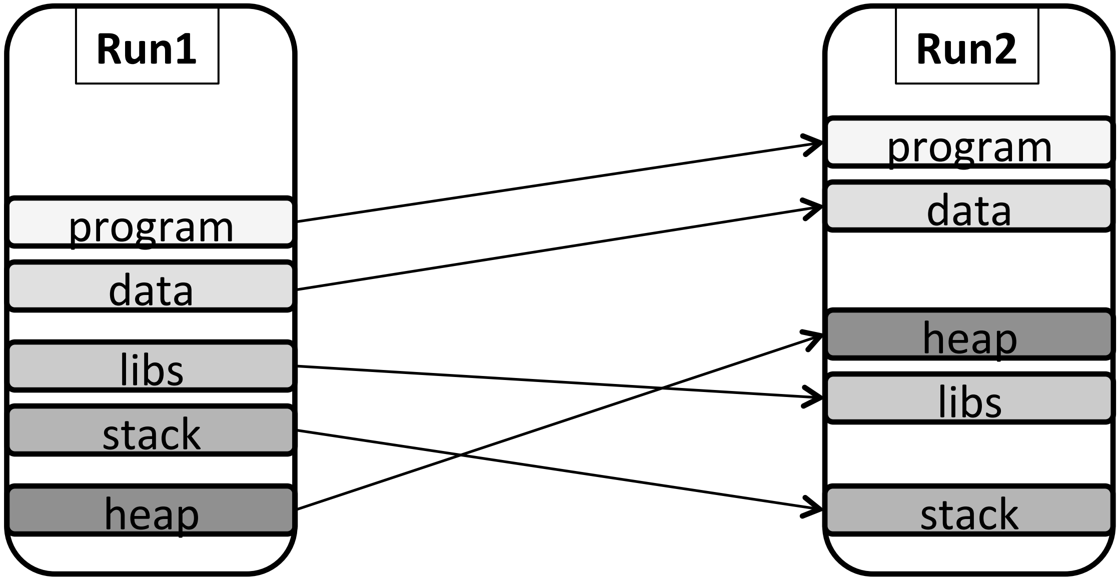
1. Knowing address of existing code gadgets

2. Overwriting control data with your address

Address space Randomizations, Re-randomizations …

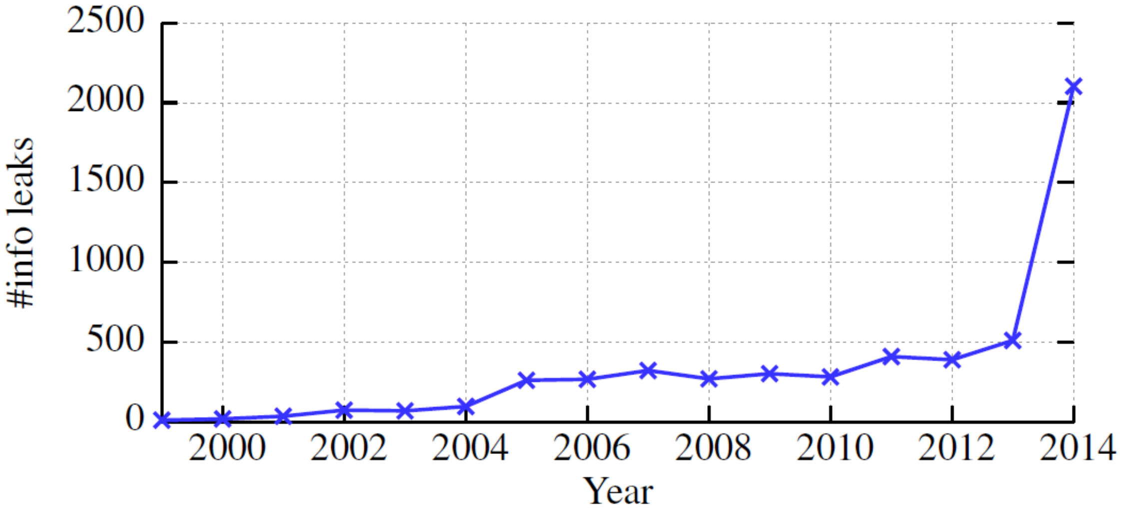
Address Space Layout Randomization (ASLR)

- Efficient, deployed in all modern OS



A Fundamental Limitation: Information Leak

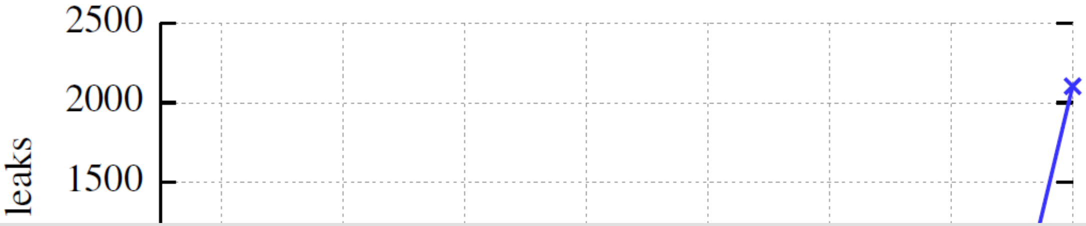
- Code pointer leak → infer code address
  - e.g., JIT-ROP, Blind ROP, “Missing the point”, etc.
- Such bugs are common, increasing!



http://www.cvedetails.com/vulnerabilities-by-types.php

A Fundamental Limitation: Information Leak

- Code pointer leak → infer code address
  - e.g., JIT-ROP, Blind ROP, “Missing the point”, etc.
- Such bugs are common, increasing!



Security guarantee of ASLR is gone!

http://www.cvedetails.com/vulnerabilities-by-types.php

Research Goal:
to prevent code pointer leaks

→ Reclaim the benefits of ASLR

Challenges

- Many ways to locate code gadgets
  - Direct: Return addr, func pointer, vtable, etc.
  - Indriect: jmp table, etc
- Code pointers are everywhere
  - Propagated as data
- Performance!

ASLR-Guard

An extremely efficient scheme
to hide or obfuscate code pointers!

Two Main Contributions

- Systematic way to discover code pointers
  - Validated with memory snapshot comparisons
- Two techniques to prevent code pointer leaks
  - Isolation
  - Encryption

Systematic Code Pointer Discovery (1)

- How are code pointers created?
  - By relocation: *loader* must relocate ALL static pointers
    - E.g., fn = base + offset
  - From program counter (PC)
    - E.g., lea offset(%rip), %rax
  - From OS
    - E.g., entry point, exception handler

Systematic Code Pointer Discovery (1)

- How are code pointers created?
  - By relocation: *loader* must relocate ALL static pointers
    - E.g., fn = base + offset
  - From program counter (PC)
    - E.g., lea offset(%rip), %rax

How to completely catch them?

Systematic Code Pointer Discovery (2)

- Relocation-based code pointers

  → Hook relocation with our custom *loader*

- PC-based code pointers

  → Complete control of toolchains (e.g., gcc, gas ...)

- OS-injected code pointers

  → Tool to scan process memory

- Data pointers?

  → They are safe as we decouple code and data

Discovered Code Pointers

No propagation

- Return address
- GOTPLT entry
- Jump table entry
- …

Propagated as data

- Base address
- Static func pointer
- Virtual func pointer
- GetPC/GetRet
- Entry point
- Exception handler
- …

More details can be found in the paper

How to protect all the discovered code pointers?

Isolation + Encryption

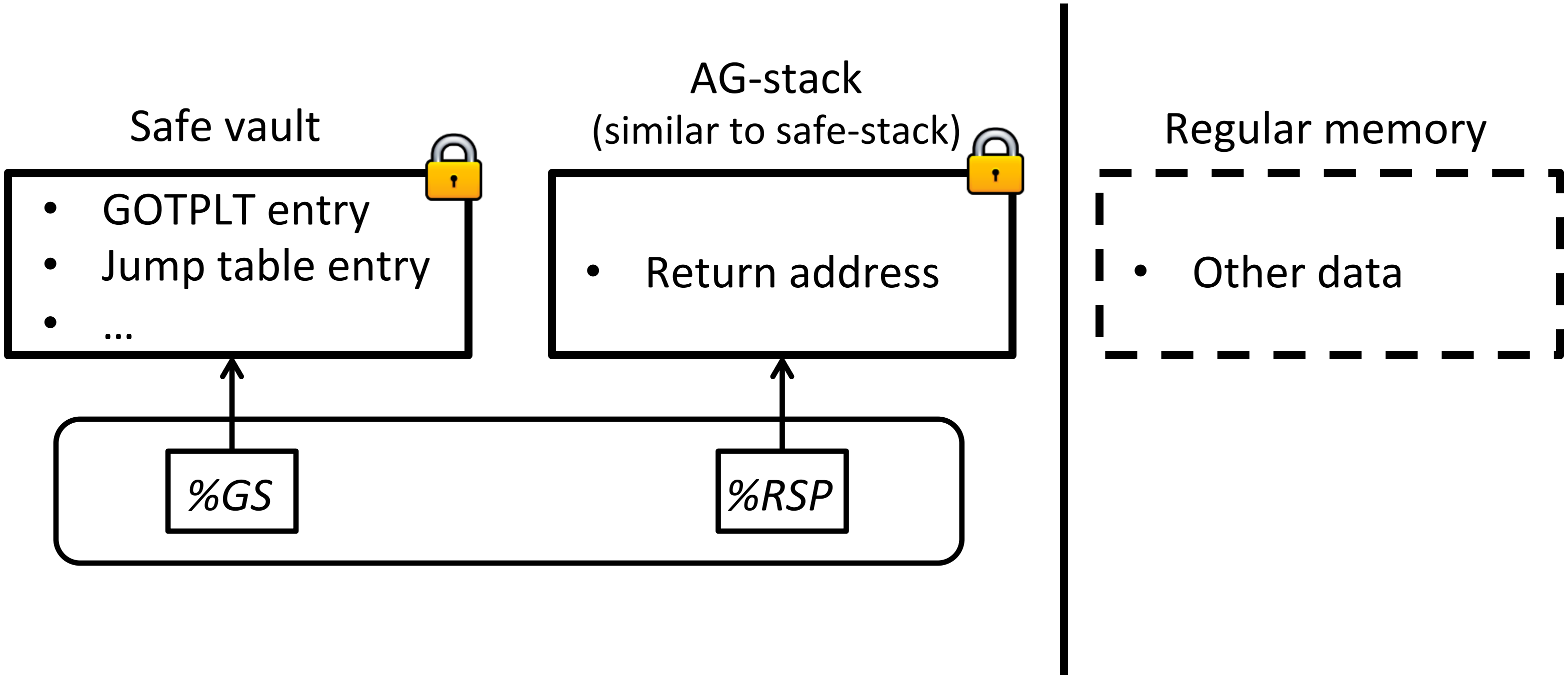
Code Pointer Isolation

- Code pointers are saved in isolated memory
  - attackers cannot touch

- Isolation is achieved by randomization (x64)
  - Fact: brute-forcingly guessing the randomized address on x64 → crash
  - Say 16 MB memory, 2^28 entropy
    - P_{hit} = 16M/(2^28 * PageSize) = 1/32,768
    - Entropy can be extended to up to 2^47

Code Pointer Isolation

- Safe vault and AG-Stack at random address
- Reserve register %GS and %RSP



Safe vault

- GOTPLT entry
- Jump table entry
- …

AG-stack
(similar to safe-stack)

- Return address

Regular memory

- Other data

Code Pointer Isolation

No propagation

- Return address
- GOTPLT entry
- Jump table entry
- …

Isolated

Propagated as data

- Base address
- Static func pointer
- Virtual func pointer
- GetPC/GetRet
- Entry point
- Exception handler
- …

Code Pointer Encryption

- When isolation is not sufficient
  - E.g., propagated to outside safe vault or AG-stack

- Three requirements
  - Confidentiality: cannot crack
  - Integrity: cannot modify
  - Efficiency

Encryption Scheme

```
void hello();
void (*fn)() = hello;
```

Assembly:

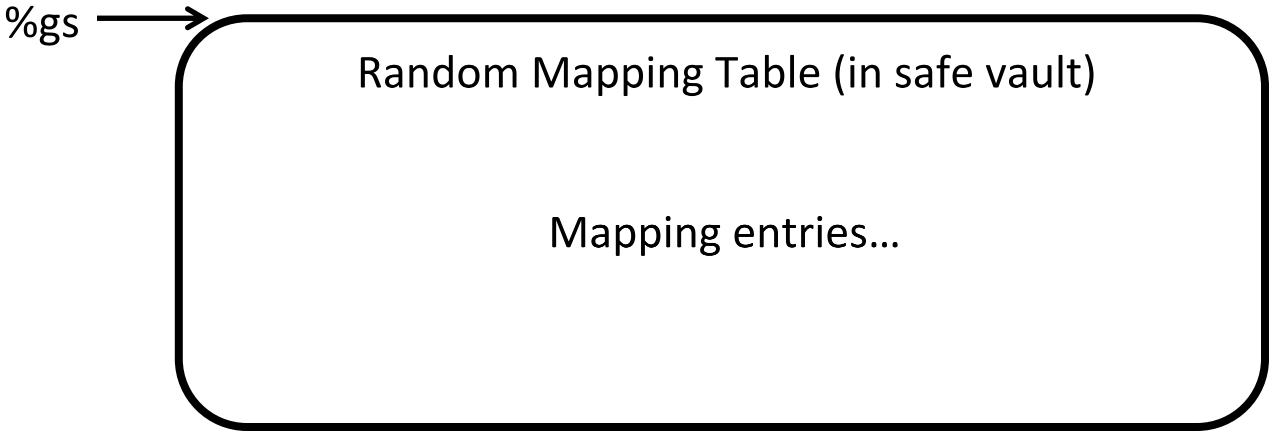
lea 0x1234(%rip), %rax

Encryption Scheme

```
void hello();
void (*fn)() = hello;
```

Assembly:

lea 0x1234(%rip), %rax



%gs

Random Mapping Table (in safe vault)

Mapping entries…

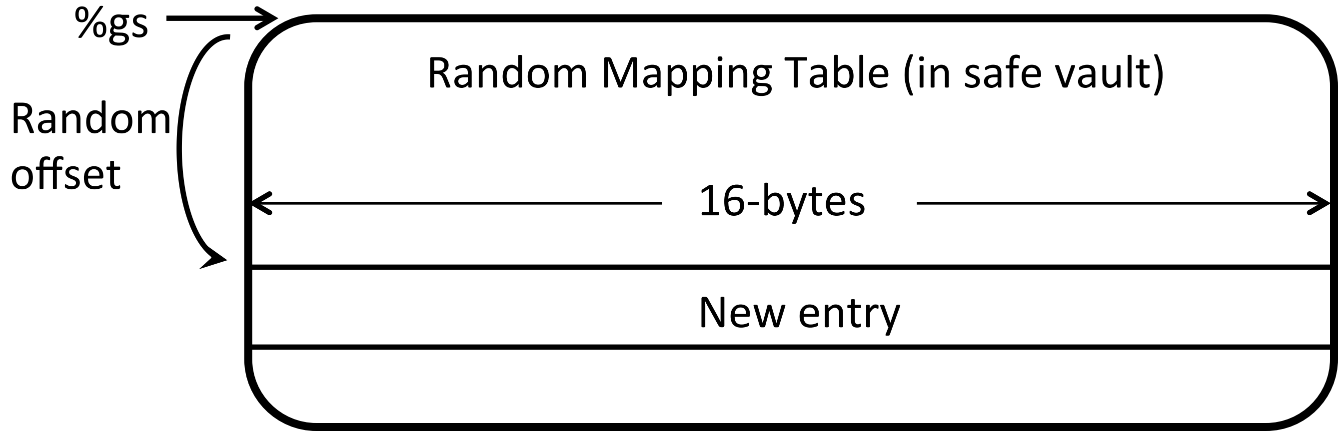
Encryption Scheme

void hello();

void (*fn)() = hello;

Assembly:

lea 0x1234(%rip), %rax



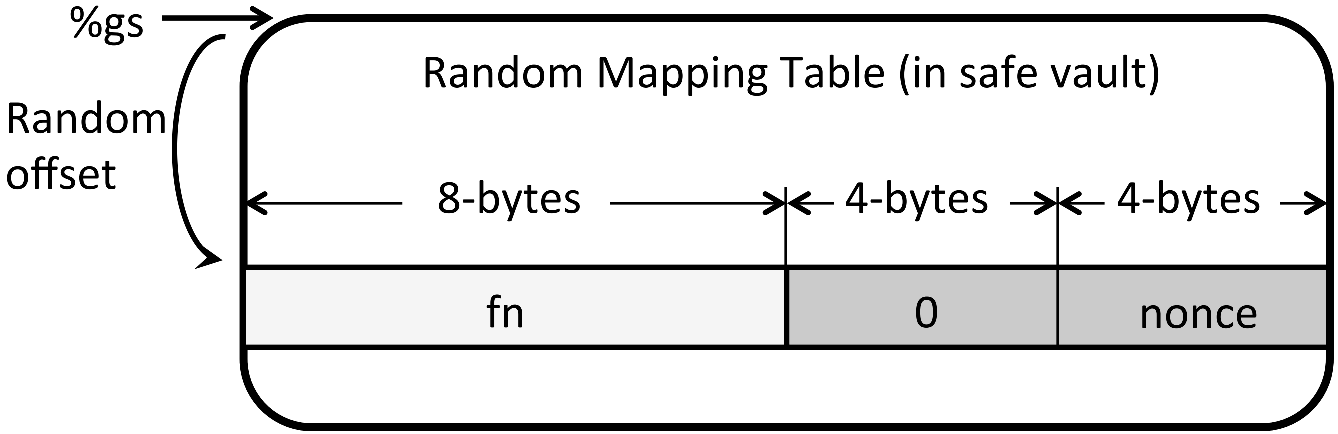
Step1: create an entry with a random offset into table base

Encryption Scheme

```
void hello();
void (*fn)() = hello;
```

Assembly:

```
lea  0x1234(%rip), %rax
```



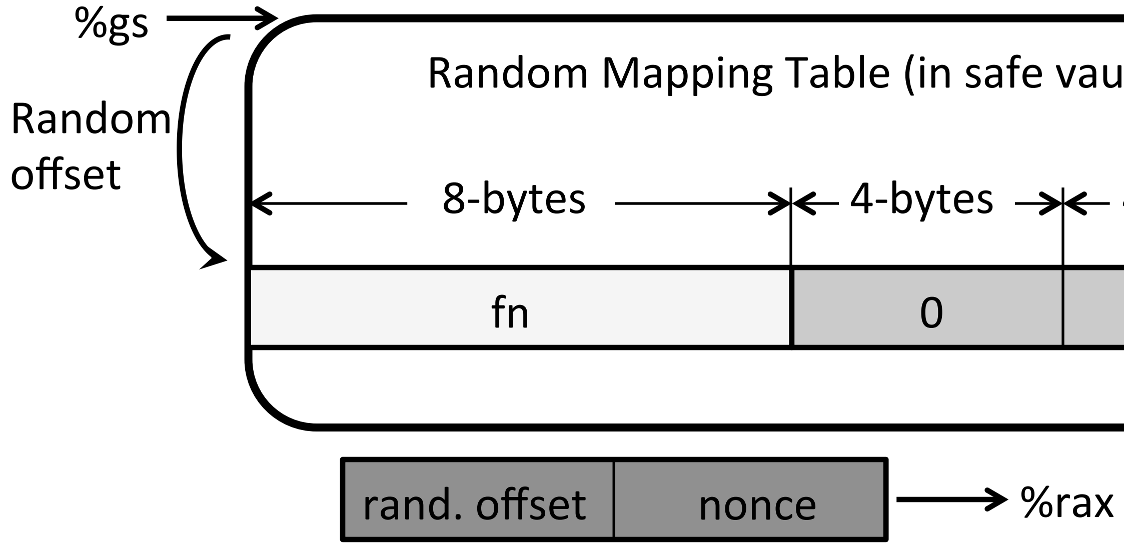
Step1: create an entry with a random offset into table base
Step2: save *fn* in first 8-bytes, followed by 4-bytes 0 and 4-bytes random nonce

Encryption Scheme

```
void hello();
void (*fn)() = hello;
```

Assembly:

lea 0x1234(%rip), %rax



%gs

Random offset

Random Mapping Table (in safe vault)

8-bytes	4-bytes	4-bytes
fn	0	nonce

rand. offset	nonce

→ %rax

Step1: create an entry with a random offset into table base

Step2: save *fn* in first 8-bytes, followed by 4-bytes 0 and 4-bytes random nonce

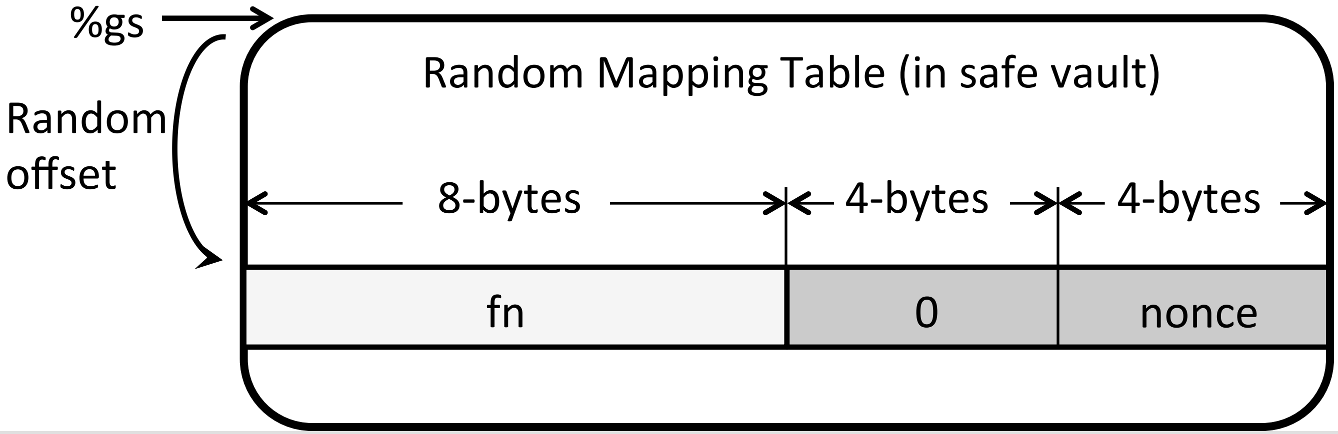
Step3: save the 4-bytes random offset and nonce into %rax

Encryption Scheme

```
void hello();
void (*fn)() = hello;
```

Assembly:

lea 0x1234(%rip), %rax



%gs

Random offset

Random Mapping Table (in safe vault)

8-bytes

4-bytes

4-bytes

fn	0	nonce

printf(“%p”, fn) →

rand. offset	nonce

Decrypt Code Pointer

fn();

Assembly:

*call *%rax;*

Decrypt Code Pointer

fn(); Assembly:

*call *%rax;*

Instrumentation :

*call *%rax;* ⟶

```
xor %gs:8(%rax), %rax;
call %gs:(%rax)
```

Decrypt Code Pointer

fn();

Assembly:

```
call *%rax;
```

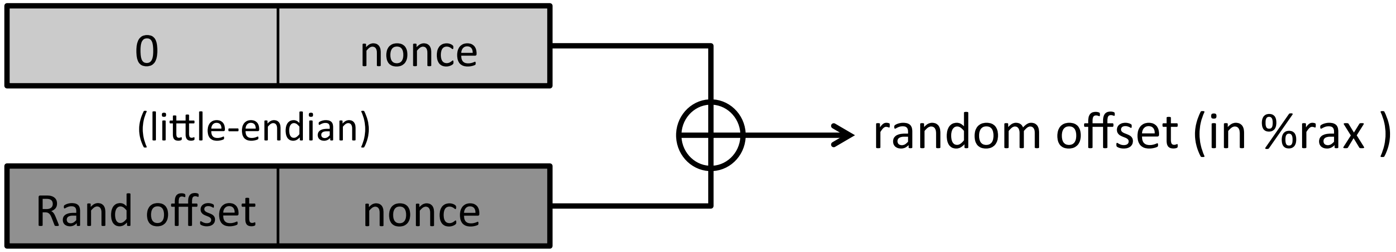
Instrumentation :

```
call *%rax;
```

→

```
xor %gs:8(%rax), %rax;
call %gs:(%rax)
```

Runtime:



0	nonce

(little-endian)

Rand offset	nonce

⊕ → random offset (in %rax)

Decrypt Code Pointer

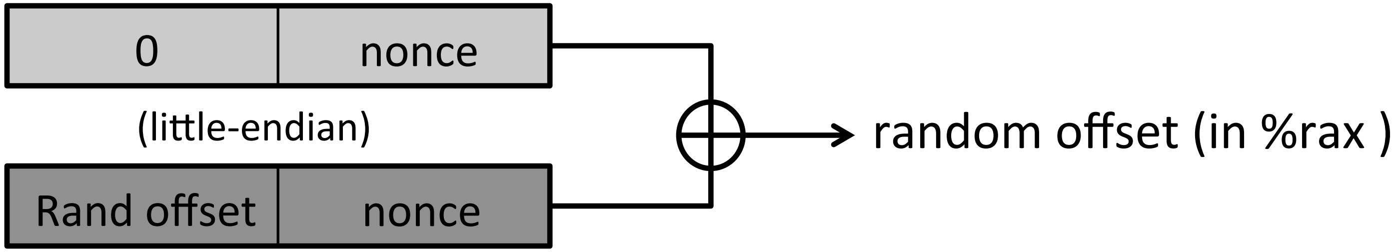
fn(); Assembly:

*call *%rax;*

Instrumentation :

*call *%rax;* → xor %gs:8(%rax), %rax;
call %gs:(%rax)

Runtime:



%gs:(%rax) points to ***"fn"*** in random mapping table,
so, call %gs:(%rax) → call fn

Decrypt Code Pointer

fn();

Assembly:

*call *%rax;*

Instrumentation :

*call *%rax;* → xor %gs:8(%rax), %rax;

call %gs:(%rax)

Runtime:

Extremely efficient decryption: only ***one XOR*** *operation!*

so, call %gs:(%rax) → call fn

More About Encryption Scheme

- It is secure
  - A secretless scheme
  - Random mapping table is isolated

- Integrity guarantee
  - Nonce per pointer
  - Single bit change → segfault (out of table)

- Secure randomness
  - Intel's RdRand instruction

Comprehensive Protection

No propagation

- Return address
- GOTPLT entry
- Jump table entry
- …

Isolated

Propagated as data

- Base address
- Static func pointer
- Virtual func pointer
- GetPC/GetRet
- Entry point
- Exception handler
- …

Encrypted

Implementation

- GNU Toolchain: gcc, gas, ld, ld.so
  - ~3000 LoC changes

- Libraries: eglibc, libstdc++ ...

- Tested on Ubuntu 14.04 X86_64 and Ubuntu 15.04 X86_64

Performance Evaluation

- <1% runtime overhead on SPEC benchmarks
- No overhead for AG-Stack
- 6% binary size increase
- >2 MB of memory overhead
- 27% load time

Security Evaluation

- Locating safe-vault/AG-Stack → 2^28
- Breaking nonce → 2^32

- Memory snapshot analysis
  - No single plain code pointer found for all SPEC benchmarks
  - No plain locator found in Nginx and blind ROP is defeated

Discussion & Limitation

- Reusing encrypted code pointers
  1) Exploiting arbitrary read
  2) Understanding semantics of leaked memory
  3) Preparing parameters

- Dynamic code generation
- DWARF exception is not implemented yet

Conclusion

- ASLR-Guard: a fast defense mechanism to prevent code pointer leaks for code reuse attacks

  → **Benefits of ASLR can be reclaimed**

Thanks!

Questions?