ROP CFI RAP XNR CPI WTF?

Navigating the Exploit Mitigation Jungle

Michael Rodler - @f0rki

2017-03-10

DEP?

Bypassing DEP?

ASLR?

Bypassing ASLR?

Bypassing Windows CFG/EMET?

So what's the problem again?

Spatial Memory Safety Violation

```
1 char array[8];
2 for (int i = 0; i <= 8; i++) {  // off-by-one error
3   array[i] = '\0';  // last loop will set array[8]
4 }
```

Listing 1: Example of a spatial memory safety violation.

Temporal Memory Safety Violation

```
1  uint32_t *p, *q;
2  char *u;
3  p = malloc(8); // allocate a uint32_t[2] array
4  // ...
5  q = p + 1;     // q references the second uint32_t
6  // ...
7  free(p);
8  u = malloc(8); // likely(p == u)
9  // ...
10 *q = ...       // Use-After-Free Bug: modifies u instead
```

Listing 2: Example of a temporal memory safety violation.

Arbitrary Code Execution?

- Overwrite code pointer
  - e.g. return address, function pointer, vtable pointer, ...
- Code injection – mitigated by DEP/NX
  - aka. shellcode
- Code reuse attacks
  - return to libc, ROP [22], JOP [5], COOP [21], ...

Full Memory Safety

- Associate **bounds** to pointer (spatial safety)
  - 64 bit pointer
  - + 64 bit upper bound
  - + 64 bit lower bound
  - e.g. Softbound [19], Intel MPX
- Associate **lifetime** information to pointer (temporal safety)
  - Dangling pointer checks
  - Double/invalid free detection
  - e.g. CETS [18]
- Propagate this through pointer arithmetic!
- **High overhead!**

The quest for low-overhead protection begins...

Let's randomize things!

ASLR

Archlinux

```
$ cat /proc/self/maps
00400000-00408000 r-xp                   /usr/bin/cat
00607000-00608000 r--p                   /usr/bin/cat
00608000-00609000 rw-p                   /usr/bin/cat
0117f000-011a0000 rw-p                   [heap]
7efe918eb000-7efe91a80000 r-xp           /usr/lib/libc-2.24.so
[...]
7ffcdefbd000-7ffcdefde000 rw-p           [stack]
[...]

$ cat /proc/self/maps
00400000-00408000 r-xp                   /usr/bin/cat
00607000-00608000 r--p                   /usr/bin/cat
00608000-00609000 rw-p                   /usr/bin/cat
009d0000-009f1000 rw-p                   [heap]
7f044c6d2000-7f044c867000 r-xp           /usr/lib/libc-2.24.so
[...]
7ffc2e798000-7ffc2e7b9000 rw-p           [stack]
[...]
```

ASLR – PIE

Fedora 25

```
$ cat /proc/self/maps
55a64db75000-55a64db81000 r-xp     /usr/bin/cat
55a64dd80000-55a64dd81000 r--p     /usr/bin/cat
55a64dd81000-55a64dd82000 rw-p     /usr/bin/cat
55a64de80000-55a64dea1000 rw-p     [heap]
7fa62929c000-7fa629459000 r-xp     /usr/lib64/libc-2.24.so
[...]
7ffe1c33e000-7ffe1c35f000 rw-p     [stack]
[...]

$ cat /proc/self/maps
556c0fdd1000-556c0fddd000 r-xp     /usr/bin/cat
556c0ffdc000-556c0ffdd000 r--p     /usr/bin/cat
556c0ffdd000-556c0ffde000 rw-p     /usr/bin/cat
556c11122000-556c11143000 rw-p     [heap]
7fd3abe04000-7fd3abfc1000 r-xp     /usr/lib64/libc-2.24.so
[...]
7ffebb896000-7ffebb8b7000 rw-p     [stack]
[...]
```

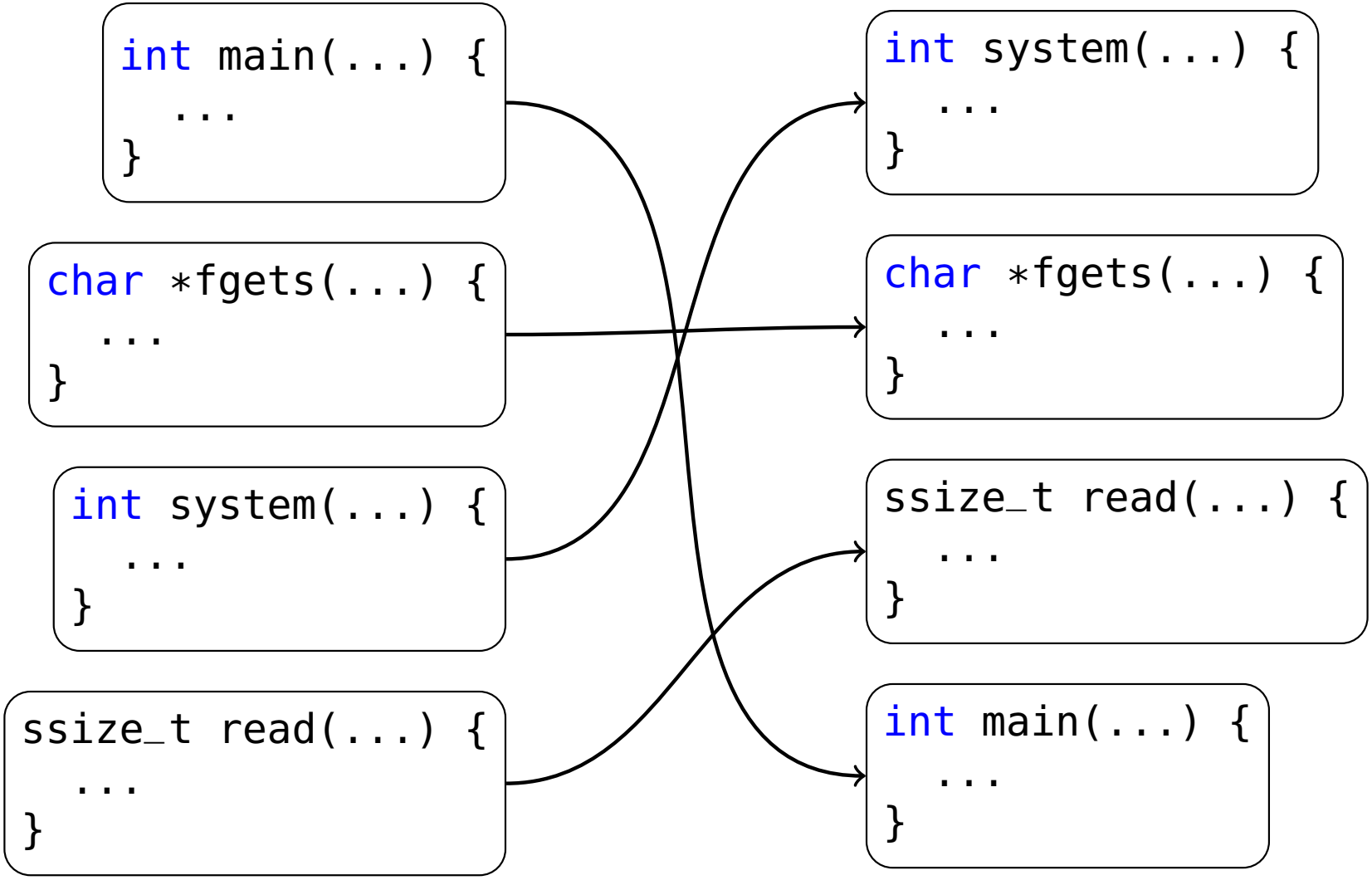
ASLR Problems

- Low entropy on non 64 bit systems
- Constant mappings for legacy reasons
- Sloppy ASLR implementations
- No-rerandomize on fork!

Let's randomize more things!

Randomize Function Order

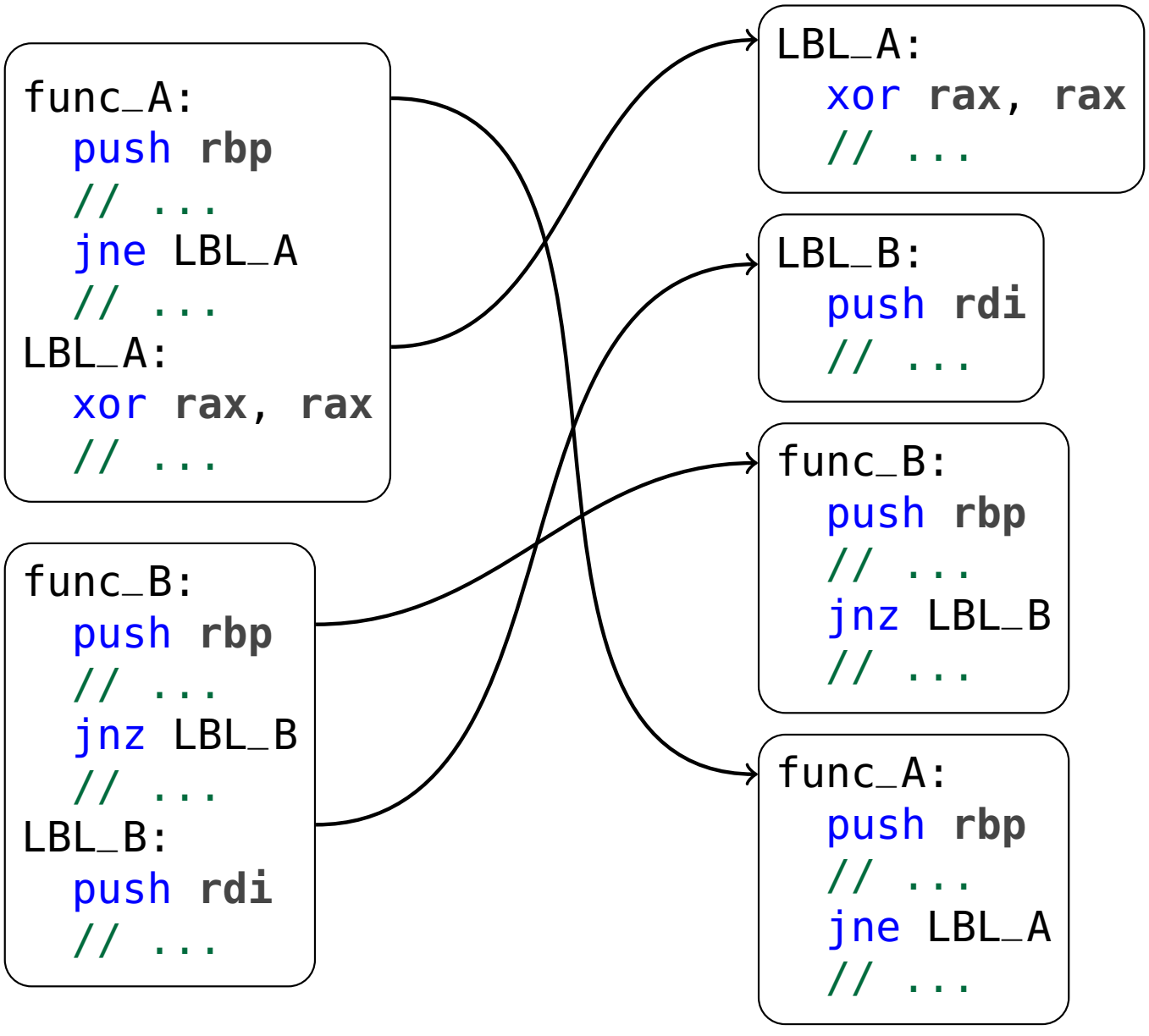
Address Space Layout Permutation [16]



Not enough entropy...

Let's randomize even more!

Permute Basic Blocks [11]



Still not enough randomness!!!

Let's randomize a little bit more!

Randomize General Purpose Register Usage

Original

```
mov eax, dword [rsp + 0x20]
movsxd rdx, eax
mov rax, qword [rsp + 0x30]
add rax, rdx
movzx eax, byte [rax]
cmp al, 0x2f
```

Variant 1

```
mov edx, dword [rsp + 0x20]
movsxd rdi, edx
mov rdx, qword [rsp + 0x30]
add rdx, rdi
movzx ebx, byte [rdx]
cmp bl, 0x2f
```

Variant 2

```
mov ecx, dword [rsp + 0x20]
movsxd rsi, ecx
mov rbx, qword [rsp + 0x30]
add rbx, rsi
movzx edx, byte [rbx]
cmp dl, 0x2f
```

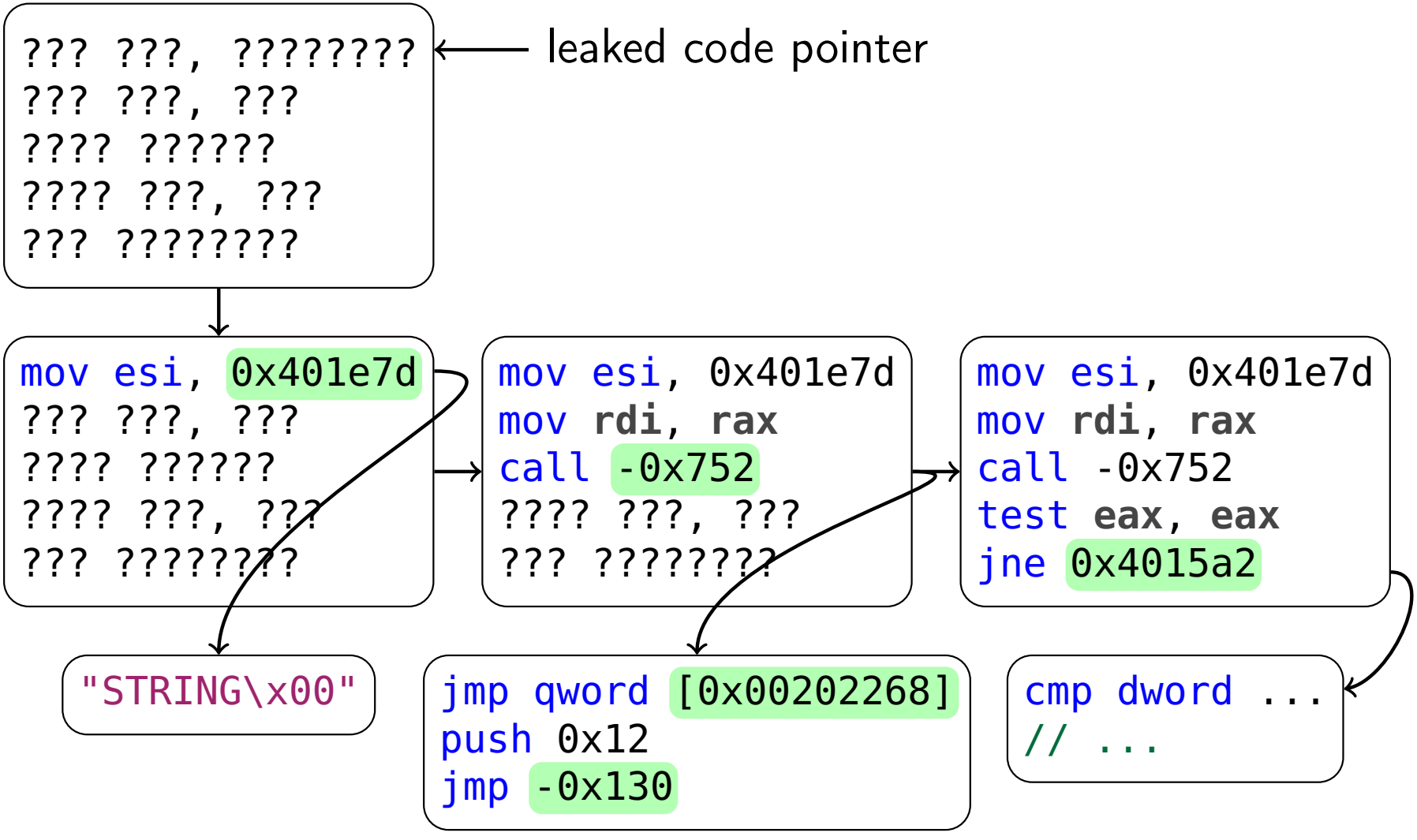
Are we good now?

JIT-ROP: Infoleak the running code!

- Systematically leak running code
- Build ROP chain "just-in-time"
- [23]

- Assumptions:
  - Arbitrary read vulnerability
  - Leak one code pointer

JIT-ROP


??? ???, ????????
??? ???, ???
???? ??????
???? ???, ???
??? ????????
leaked code pointer
mov esi, 0x401e7d
??? ???, ???
???? ??????
???? ???, ???
??? ????????
mov esi, 0x401e7d
mov rdi, rax
call -0x752
???? ???, ???
??? ????????
mov esi, 0x401e7d
mov rdi, rax
call -0x752
test eax, eax
jne 0x4015a2
"STRING\x00"
jmp qword [0x00202268]
push 0x12
jmp -0x130
cmp dword ...
// ...

Blind ROP

- Search for infoleak ROP chain by trial & crash
- This is totally feasible! [4]
- Assumptions
  - Unlimited tries (e.g. fork) or crash-resistance (e.g. [14])
  - No re-randomization
  - Need to know rough code location

Leaking Code Pointers

- Great way to defeat fine-grained randomization
- Function-level code-reuse of leaked function pointers
  - ret2libc is Turing-complete [25]
  - COOP attacks: virtual method-level code-reuse
    - Leak vtable pointers
    - Chain C++ virtual method calls

Well...

Let's stop infoleaks?!?

Mitigate Code Infoleaks

- Multivariate Execution (MVX, e.g. Detile) [13]
- Prevent/Mitigate code disclosure
  - Execute-no-read (XnR) [3]
  - Destructive code-reads (DCR, e.g. Heisenbyte) [24]
  - No-execute-after-read (NEAR) [26]
- Hide code pointers and pointers to code pointers
  - Leak vtables, jump tables, etc.
  - e.g. readactor(++) [9, 8]
- Re-randomize every N ms (Shuffler) [27]
- Execution-path randomization (Isomeron) [10]

OK... Randomization is hard...

..and we didn't even talk about side-channel attacks!

What about deterministic protections?

Control-Flow Integrity [1] I

- Enforce that control-flow transfers are within CFG
- Static control-flow graph = security policy
- Static CFG over-approximate

- Indirect variable CF transfers: `jmp, call, ret`

Control-Flow Integrity [1] II

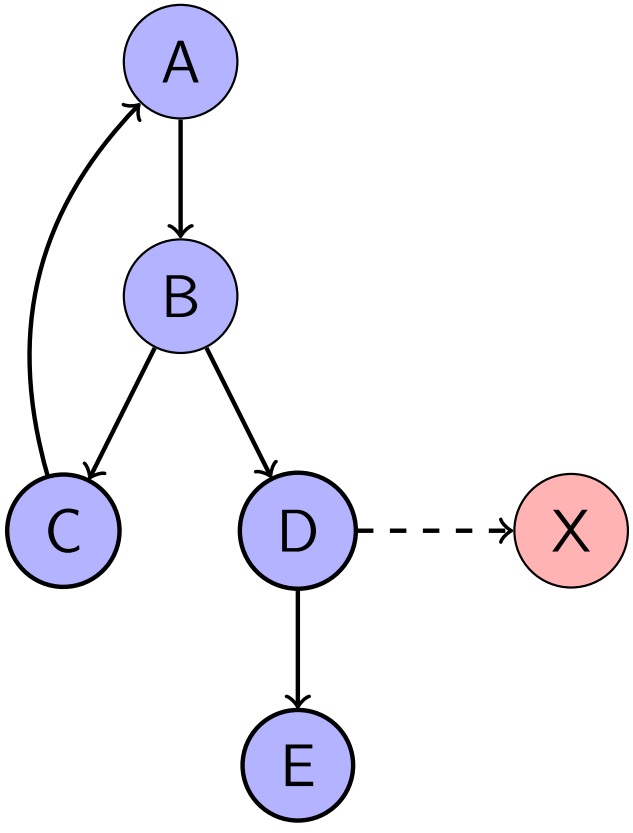


Figure: Example of control-flow graph on basic-block granularity. A control flow transfer from D to X is not intended and would violate CFI.

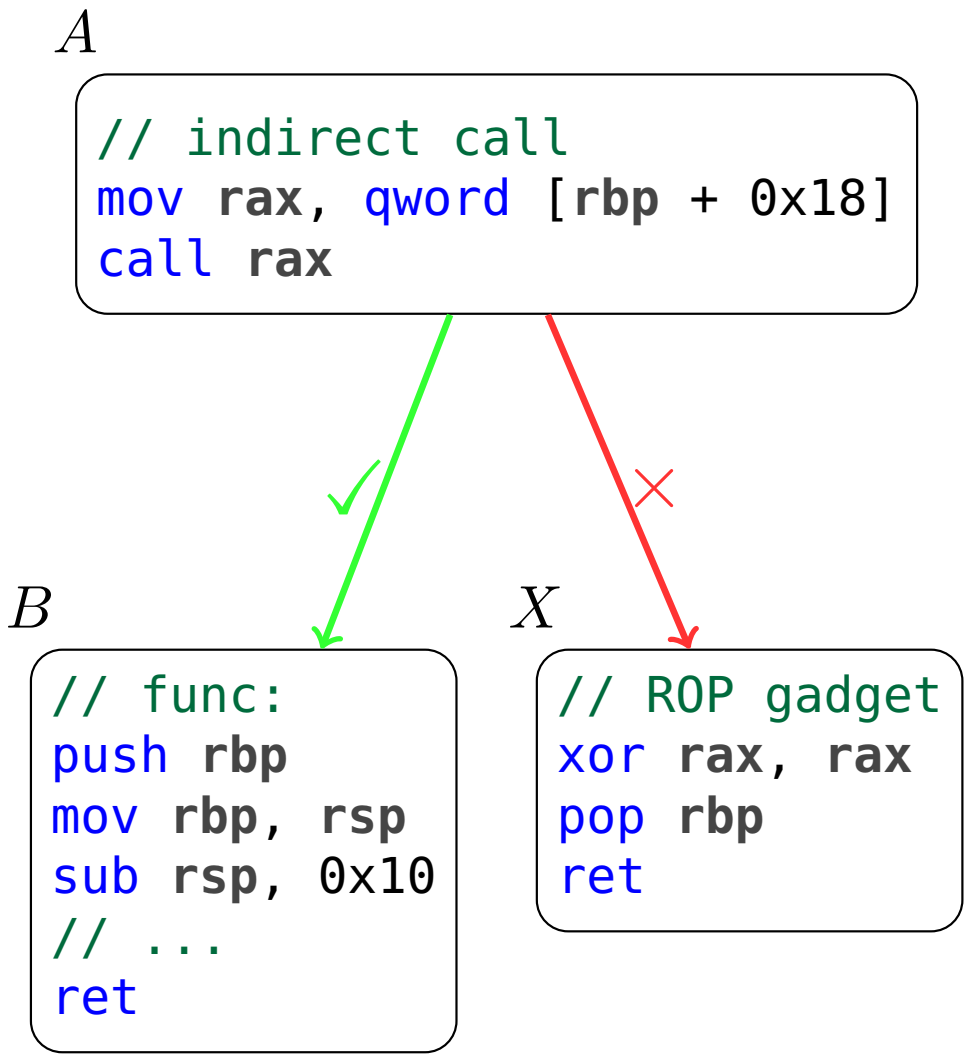


Figure: Forward-edge CFI: On the left a valid transfer and on the right a malicious one.

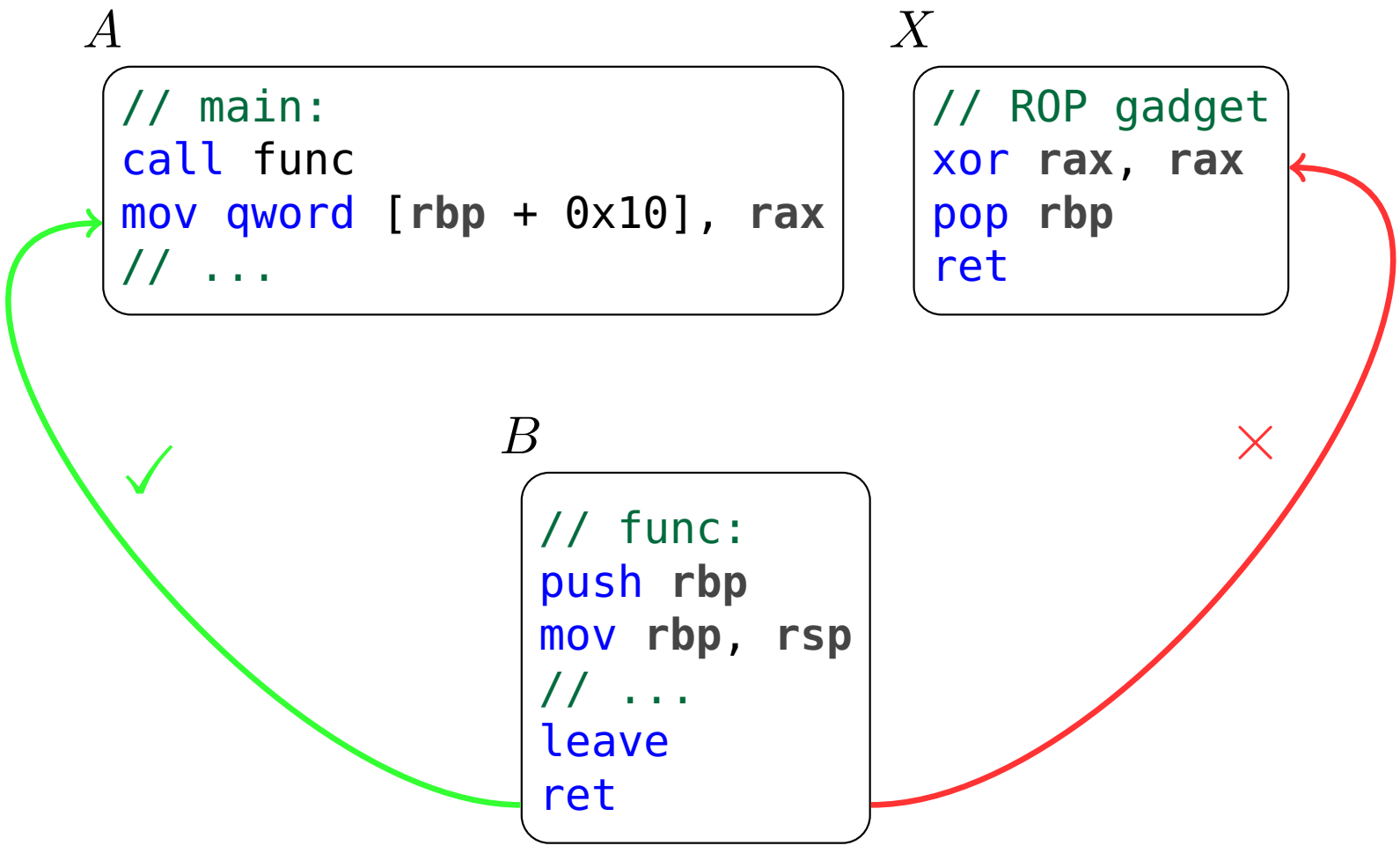


Figure: Backward-edge CFI: On the left a valid return and on the right a malicious one.

Coarse-grained CFI policies

- Return address must be preceded by `call`
- Indirect calls only to
  - Function beginnings
  - Address-taken functions
- Type-based policies for indirect calls
  - e.g. PaX RAP, LLVM CFI

```c
typedef struct msg {
  char str[64];
  int (*print_fn)(char*);
} msg_t;

int execve(char*, char**, char**) { /* ... */ }
int system(char* cmd) { /* ... */ }
int log_stdout(char* s) { /* ... */  }
int log_file(char* s) { /* ... */ }

int main(int argc, char* argv[]) {
  // ...
  msg_t msg = { .msg = {0, }, .print_fn = log_stdout};
  // this is a buggy length check.
  size_t isz = strlen(user_inp) + 1;
  // corrupt the function pointer msg.print_fn
  memcpy(msg.msg,
         user_inp,  /* <-- attacker controlled */
         isz < sizeof(msg.msg) ? isz : sizeof(msg));
  // control-flow hijack
  msg.print_fn(msg.str);
  // ...
}
```

Effectiveness of Control-Flow Integrity

- Fine-grained CFI generally too expensive
- Coarse-grained CFI implementations can be bypassed
- Theoretical perfect static CFI can be broken
  - Control-Flow Bending [6]
  - Traversing only the valid CF graph is enough
  - Functions like `memcpy` make the CFG rather dense

Oh no... What now?

Code Pointer Protection

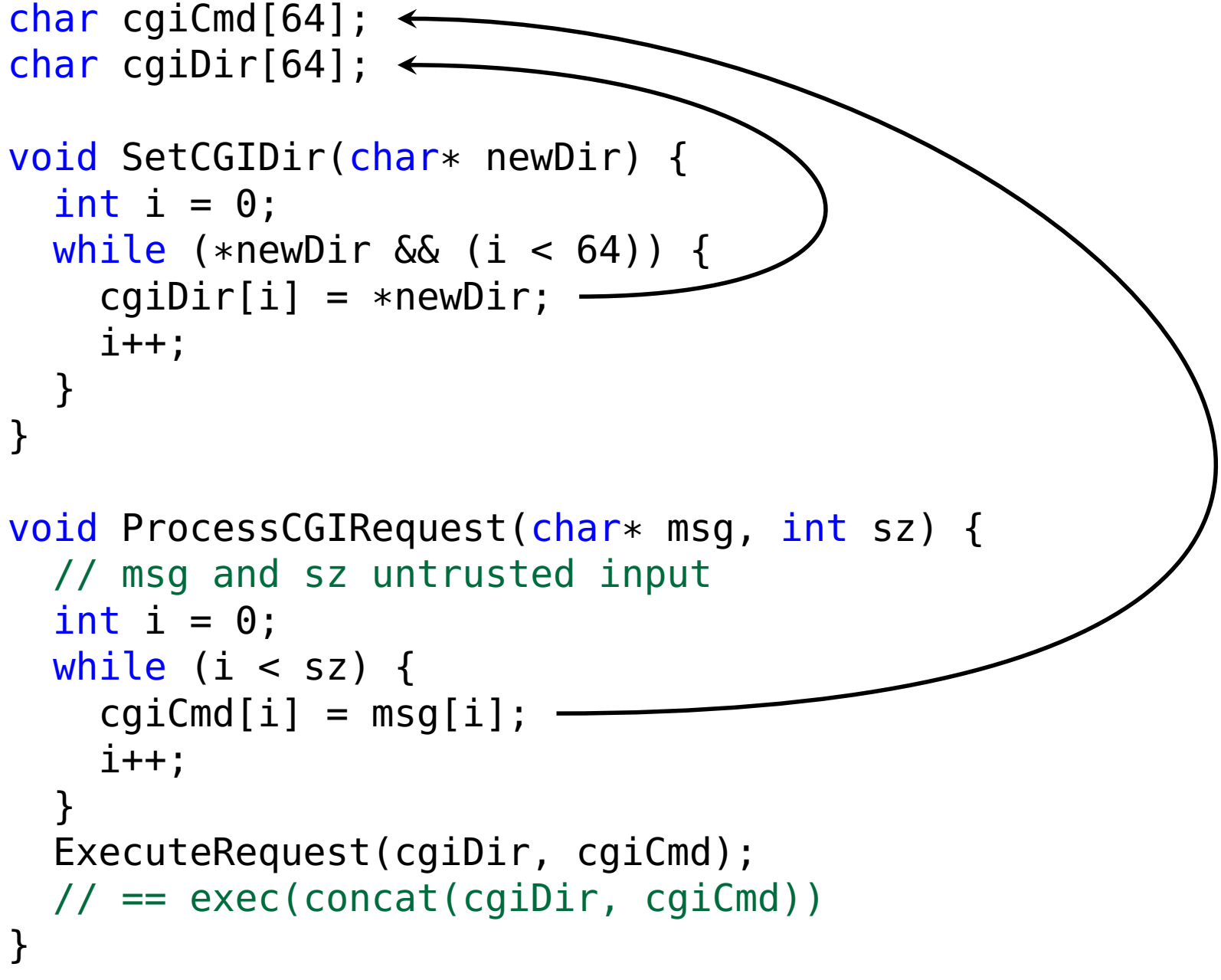
- Certain code pointers (e.g. stack canaries, RELRO)
- C++ vtable protections (e.g. [20])
- Code Pointer Integrity [17]
  - Memory safety for all code pointers
  - ...and all pointers to code pointers
- Protecting return addresses
  - High performance demands
  - Shadowstack (e.g. Windows Return Flow Guard)
  - Splitstack (e.g. LLVM SafeStack)
  - Obfuscation (e.g. PaX RAP)

Data-Only Attacks

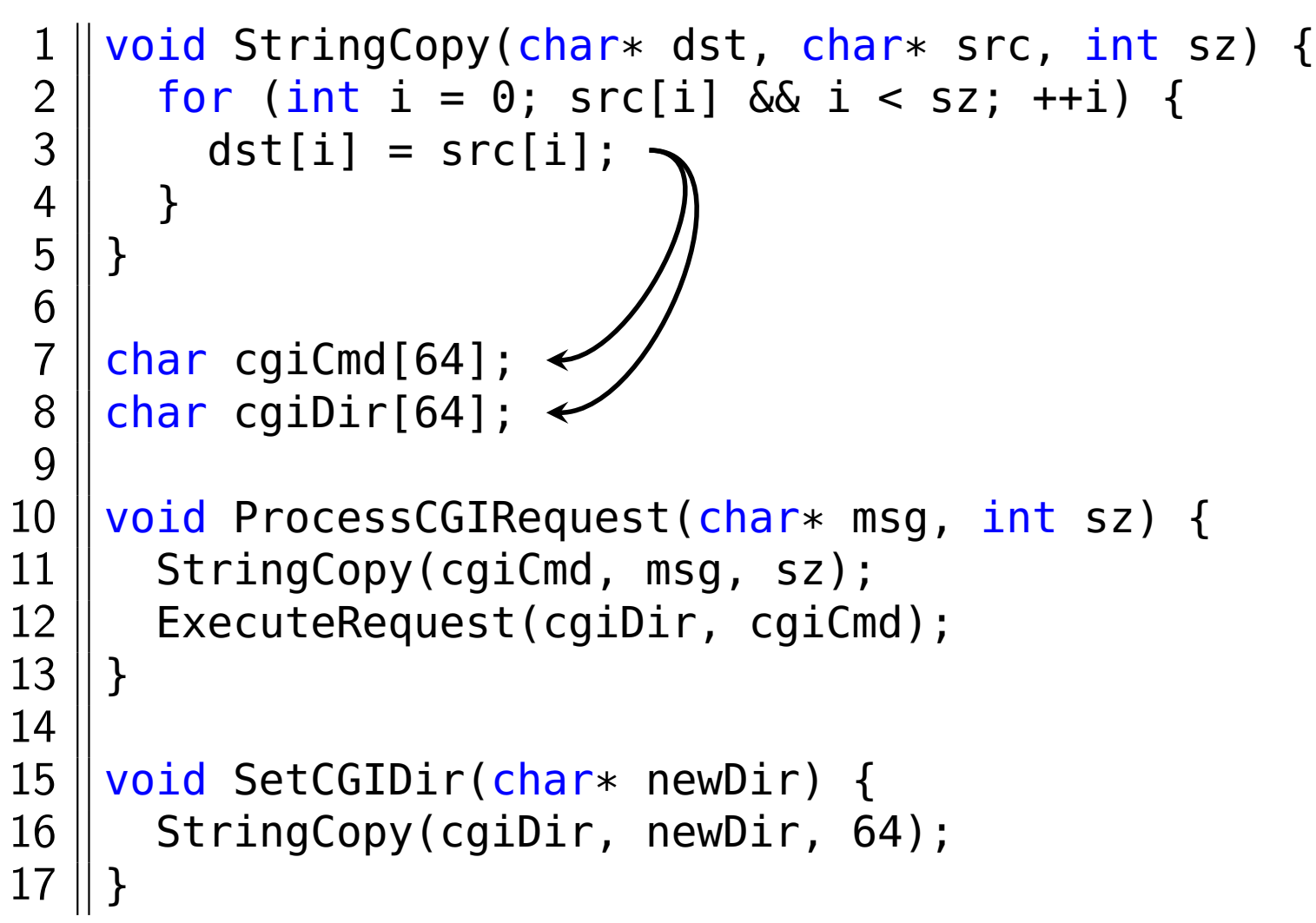
- Corrupt data variable and/or data pointers
- Trigger traverse to interesting node in CFG e.g.
  - Corrupt permission bits to elevate privileges
  - Modify money value in banking app
- Data-oriented programming (DOP) is Turing-complete [15]
- Used to break other defenses [12, 14]

Data-Flow Integrity

- Restrict which instructions can read/write which objects
- Points-to analysis = security policy
- DFI [7], Write-Integrity Testing (WIT) [2]



```
1  char cgiCmd[64];
2  char cgiDir[64];
3
4  void SetCGIDir(char* newDir) {
5    int i = 0;
6    while (*newDir && (i < 64)) {
7      cgiDir[i] = *newDir;
8      i++;
9    }
10 }
11
12 void ProcessCGIRequest(char* msg, int sz) {
13   // msg and sz untrusted input
14   int i = 0;
15   while (i < sz) {
16     cgiCmd[i] = msg[i];
17     i++;
18   }
19   ExecuteRequest(cgiDir, cgiCmd);
20   // == exec(concat(cgiDir, cgiCmd))
21 }
```


1 void StringCopy(char* dst, char* src, int sz) {
2 for (int i = 0; src[i] && i < sz; ++i) {
3 dst[i] = src[i];
4 }
5 }
6
7 char cgiCmd[64];
8 char cgiDir[64];
9
10 void ProcessCGIRequest(char* msg, int sz) {
11 StringCopy(cgiCmd, msg, sz);
12 ExecuteRequest(cgiDir, cgiCmd);
13 }
14
15 void SetCGIDir(char* newDir) {
16 StringCopy(cgiDir, newDir, 64);
17 }

Write-Integrity Testing

- Enforce data-flow integrity for store instructions
- Restricts data-only attacks

- Use shadow memory to store object sets
- Instrument every store
  - with an additional load, compare, branch

Conclusion

- Security vs performance tradeoff
- NX/ASLR made exploit dev more difficult
- CFI will make it even more difficult
- Low-overhead specialized mitigations
  - e.g. hardened allocators
  - Safestack

References I


Martin Abadi et al. “Control-flow integrity”. In: Proceedings of the 12th ACM conference on Computer and communications security. ACM. 2005, pp. 340–353.


Periklis Akritidis et al. “Preventing memory error exploits with WIT”. In: Security and Privacy, 2008. SP 2008. IEEE Symposium on. IEEE. 2008, pp. 263–277.


Michael Backes et al. “You can run but you can’t read: Preventing disclosure exploits in executable code”. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM. 2014, pp. 1342–1353.


Andrea Bittau et al. “Hacking blind”. In: Security and Privacy (SP), 2014 IEEE Symposium on. IEEE. 2014, pp. 227–242.

References II

Tyler Bletsch et al. "Jump-oriented programming: a new class of code-reuse attack". In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM. 2011, pp. 30–40.

Nicolas Carlini et al. "Control-flow bending: On the effectiveness of control-flow integrity". In: 24th USENIX Security Symposium, USENIX Sec. 2015.

Miguel Castro, Manuel Costa, and Tim Harris. "Securing software by enforcing data-flow integrity". In: Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association. 2006, pp. 147–160.

Stephen J Crane et al. "It's a TRaP: Table randomization and protection against function-reuse attacks". In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM. 2015, pp. 243–255.

References III


Stephen Crane et al. “Readactor: Practical code randomization resilient to memory disclosure”. In: Security and Privacy (SP), 2015 IEEE Symposium on. IEEE. 2015, pp. 763–780.


Lucas Davi et al. “Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming.” In: NDSS. 2015.


Lucas Davi et al. “XIFER: a software diversity tool against code-reuse attacks”. In: 4th ACM International Workshop on Wireless of the Students, by the Students, for the Students (S3 2012). Vol. 174. 2012.


Isaac Evans et al. “Missing the Point (er): On the Effectiveness of Code Pointer Integrity1”. In: IEEE Symp. on Security and Privacy. 2015.

References IV


Robert Gawlik et al. "Detile: Fine-grained information leak detection in script engines". In: Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2016, pp. 322–342.


Robert Gawlik et al. "Enabling client-side crash-resistance to overcome diversification and information hiding". In: Symposium on Network and Distributed System Security (NDSS). 2016.


Hong Hu et al. "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks". In: (2016).


Chongkyung Kil et al. "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software". In: Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual. IEEE. 2006, pp. 339–348.

References V


Volodymyr Kuznetsov et al. “Code-pointer integrity”. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2014.


Santosh Nagarakatte et al. “CETS: compiler enforced temporal safety for C”. In: ACM Sigplan Notices. Vol. 45. 8. ACM. 2010, pp. 31–40.


Santosh Nagarakatte et al. “SoftBound: highly compatible and complete spatial memory safety for c”. In: ACM Sigplan Notices. Vol. 44. 6. ACM. 2009, pp. 245–258.


Aravind Prakash, Xunchao Hu, and Heng Yin. “vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries.” In: NDSS. 2015.

References VI

Felix Schuster et al. "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications". In: Security and Privacy (SP), 2015 IEEE Symposium on. IEEE. 2015, pp. 745–762.

Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: Proceedings of the 14th ACM conference on Computer and communications security. ACM. 2007, pp. 552–561.

Kevin Z Snow et al. "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization". In: Security and Privacy (SP), 2013 IEEE Symposium on. IEEE. 2013, pp. 574–588.

References VII

Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads". In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM. 2015, pp. 256–267.

Minh Tran et al. "On the expressiveness of return-into-libc attacks". In: International Workshop on Recent Advances in Intrusion Detection. Springer. 2011, pp. 121–141.

Jan Werner et al. "No-execute-after-read: Preventing code disclosure in commodity software". In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. ACM. 2016, pp. 35–46.

References VIII

David Williams-King et al. “Shuffler: Fast and deployable continuous code re-randomization”. In: Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation. USENIX Association. 2016, pp. 367–382.