Too Subtle to Notice: Investigating Executable Stack Issues in Linux Systems

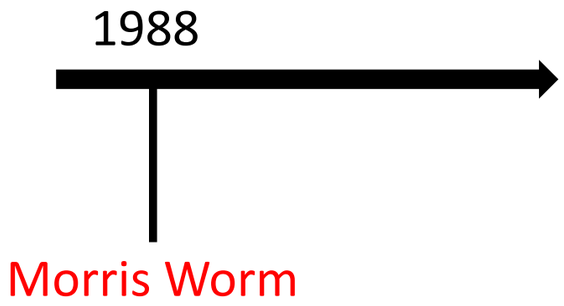
Hengkai Ye Hong Hu

History of Code Injection

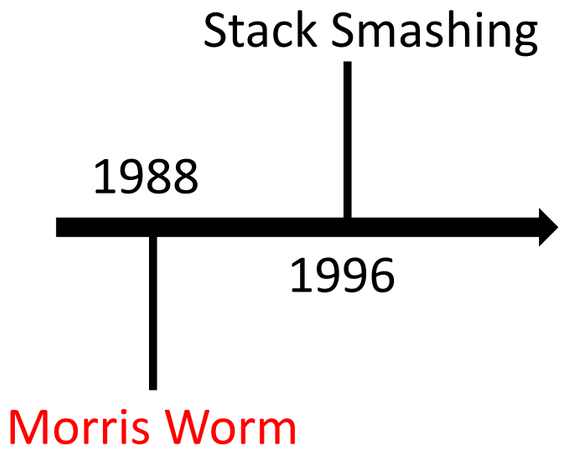
History of Code Injection

1988

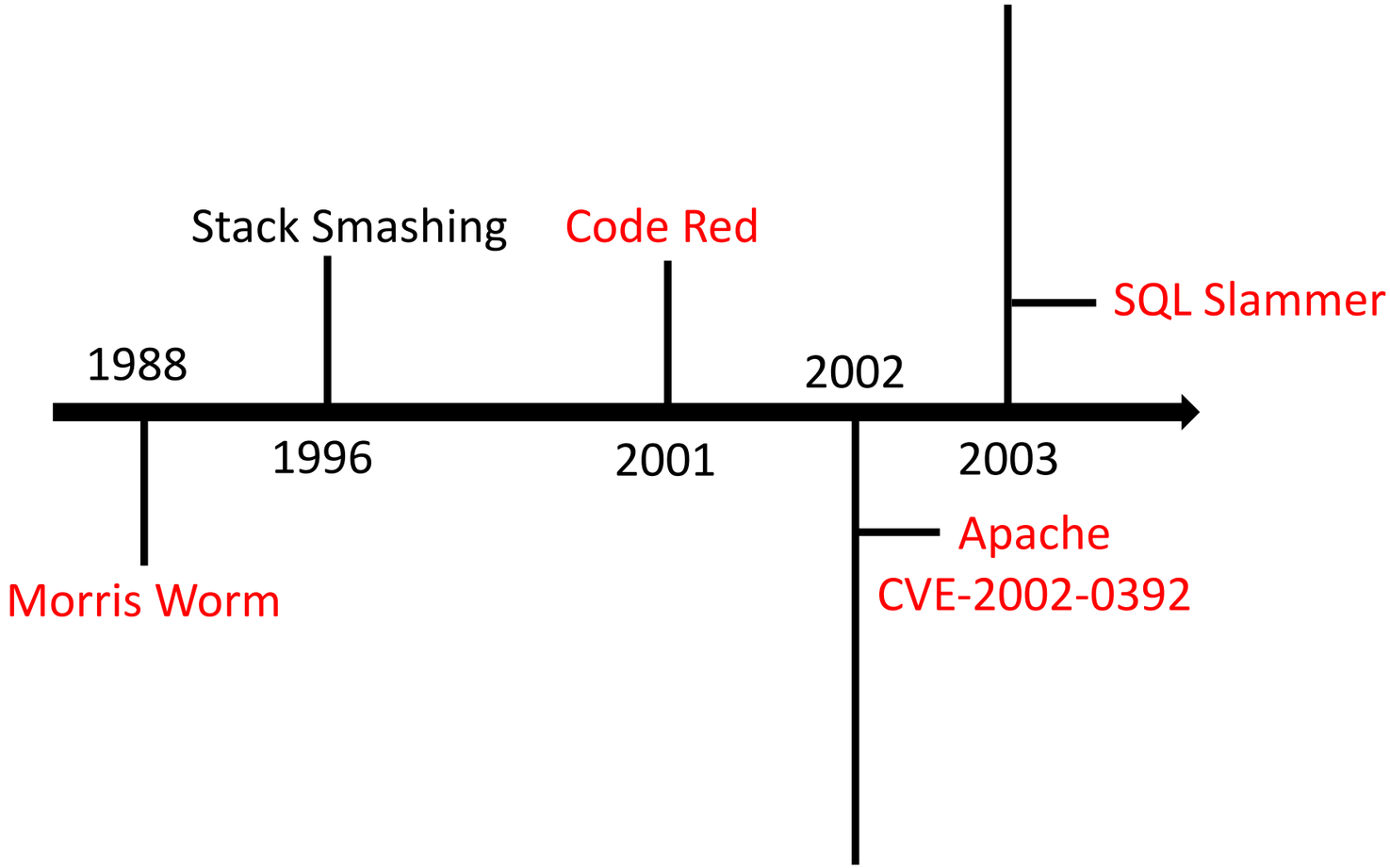
History of Code Injection



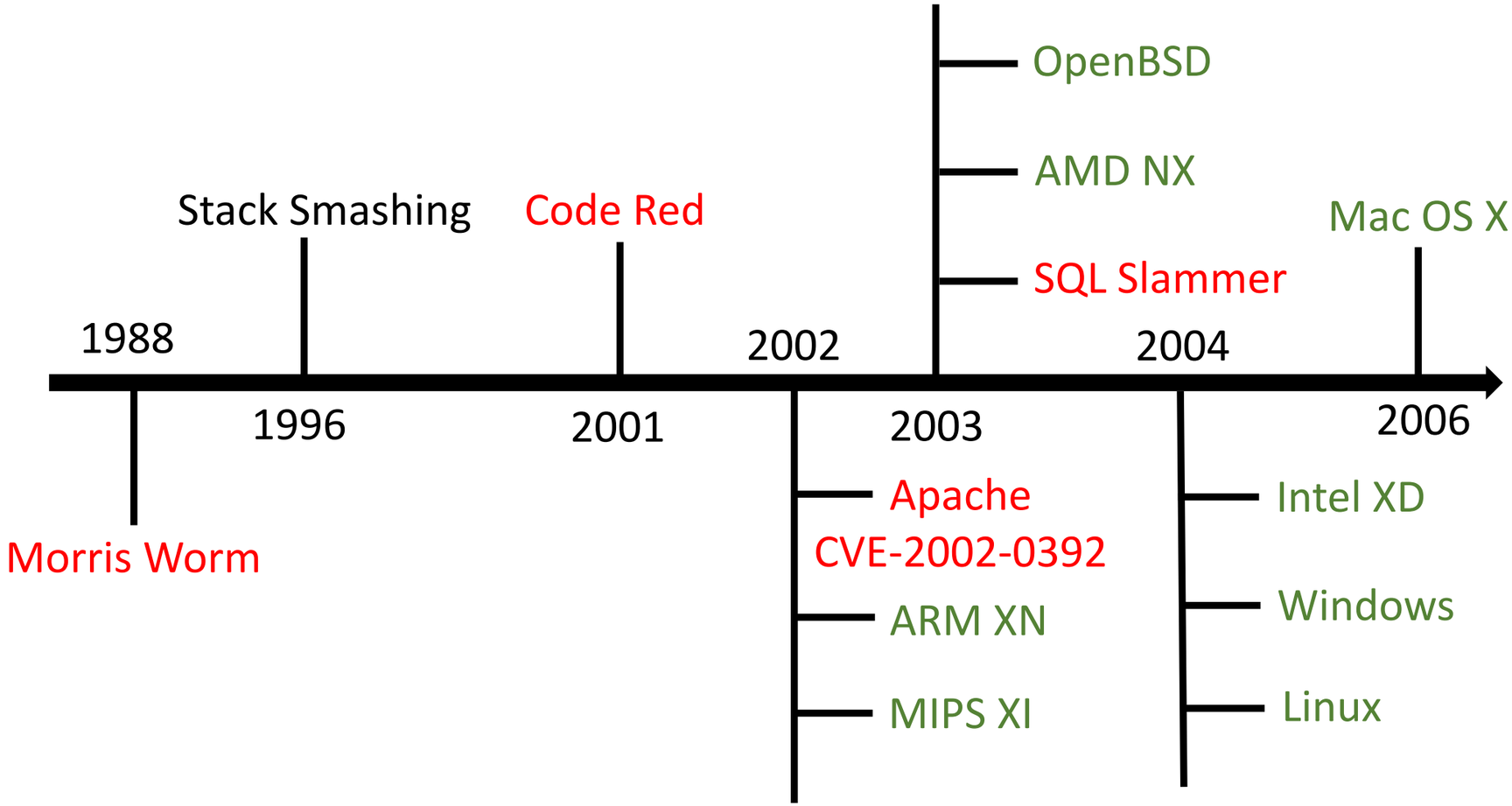
History of Code Injection



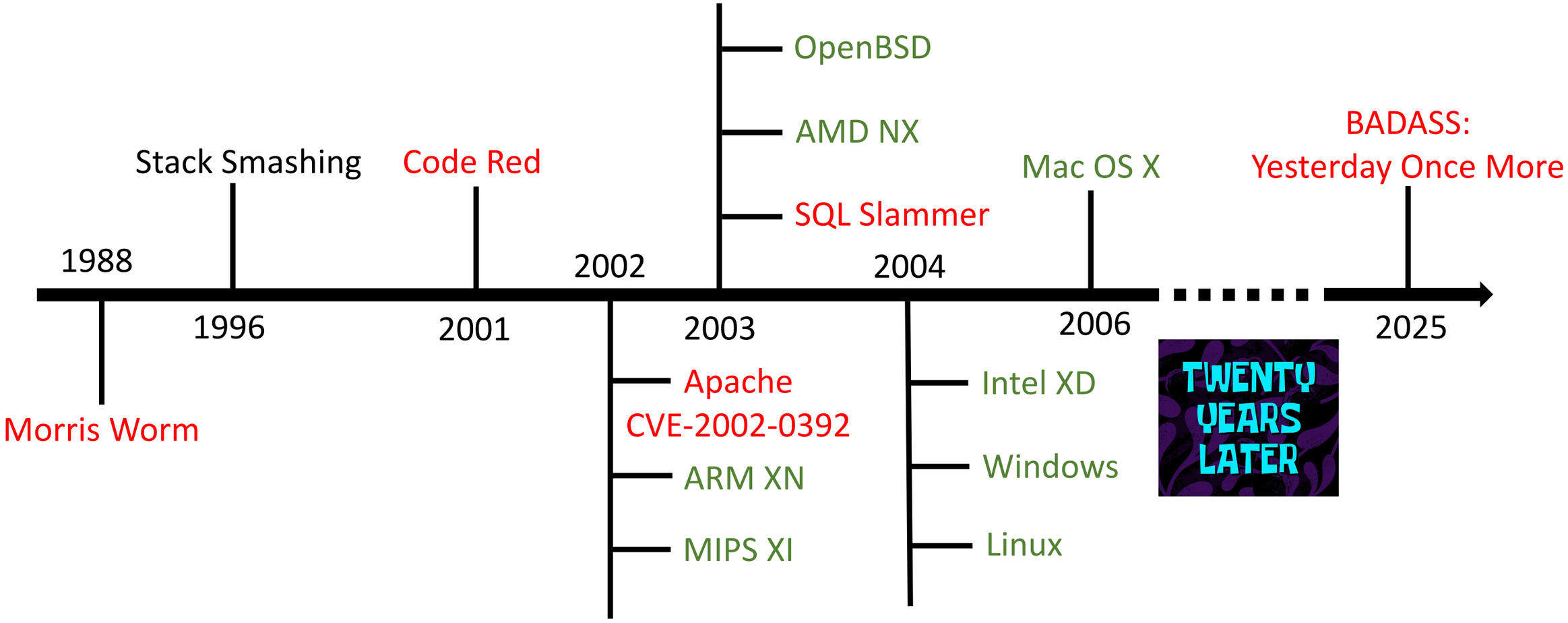
History of Code Injection



History of Code Injection



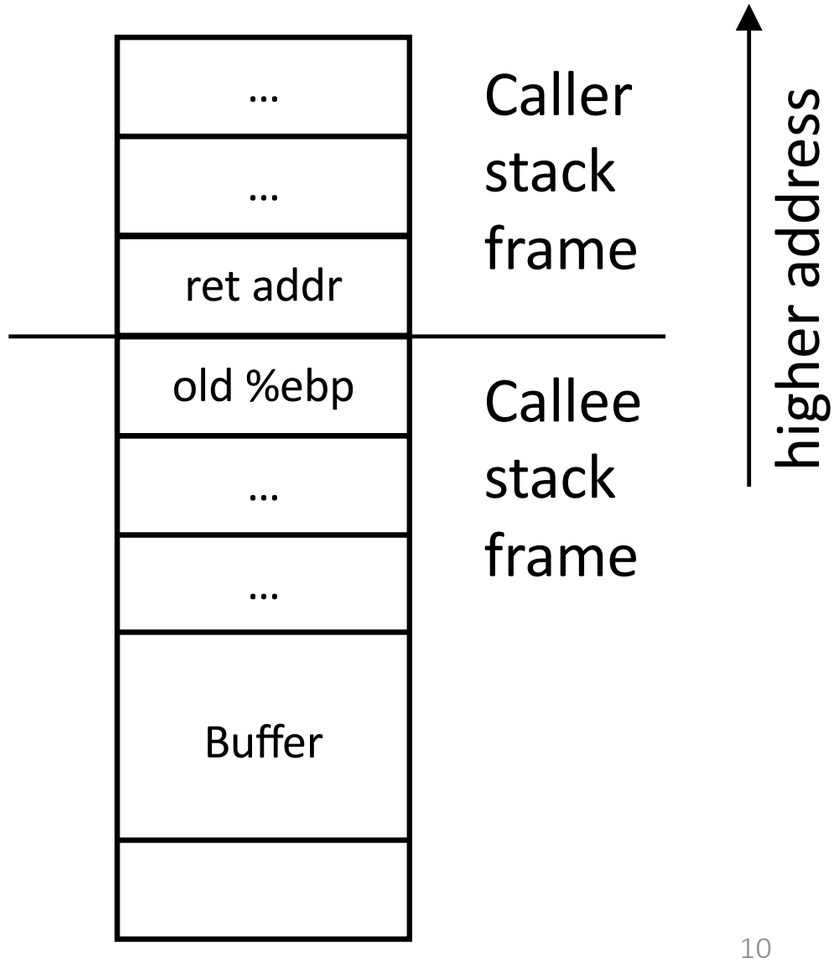
History of Code Injection



Code Injection w/ An Executable Stack 😈

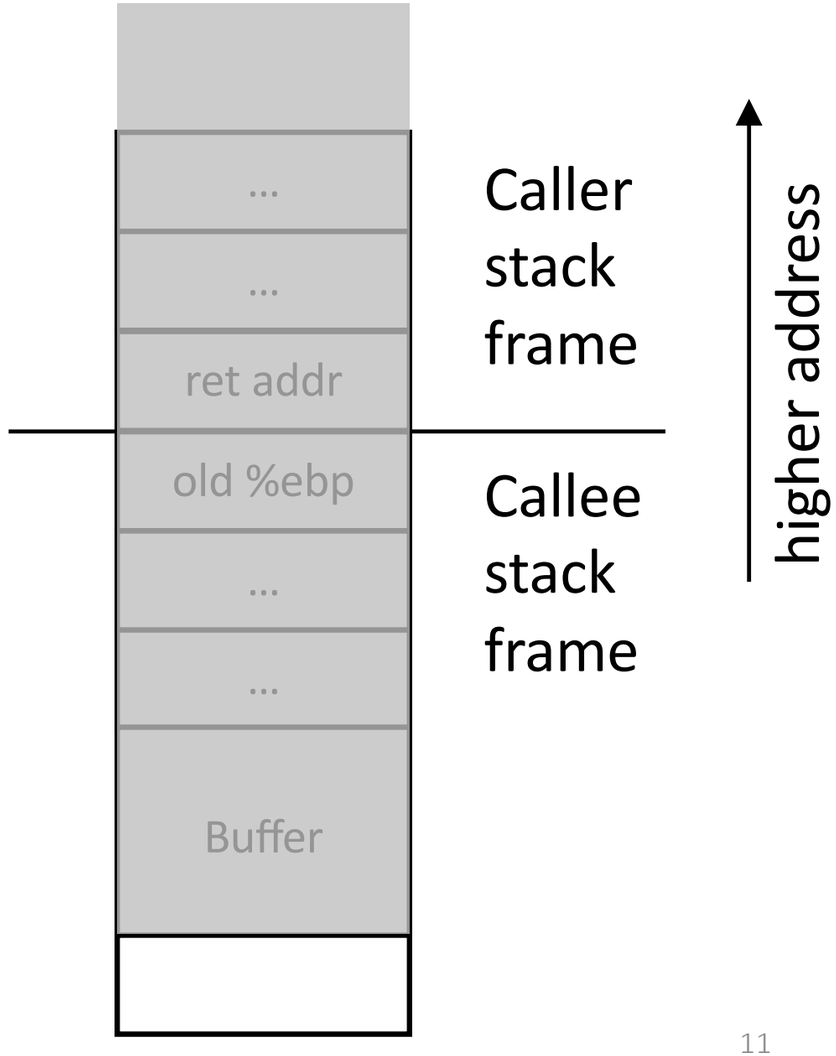
Code Injection w/ An Executable Stack 😈

- Stack buffer overflow vulnerability



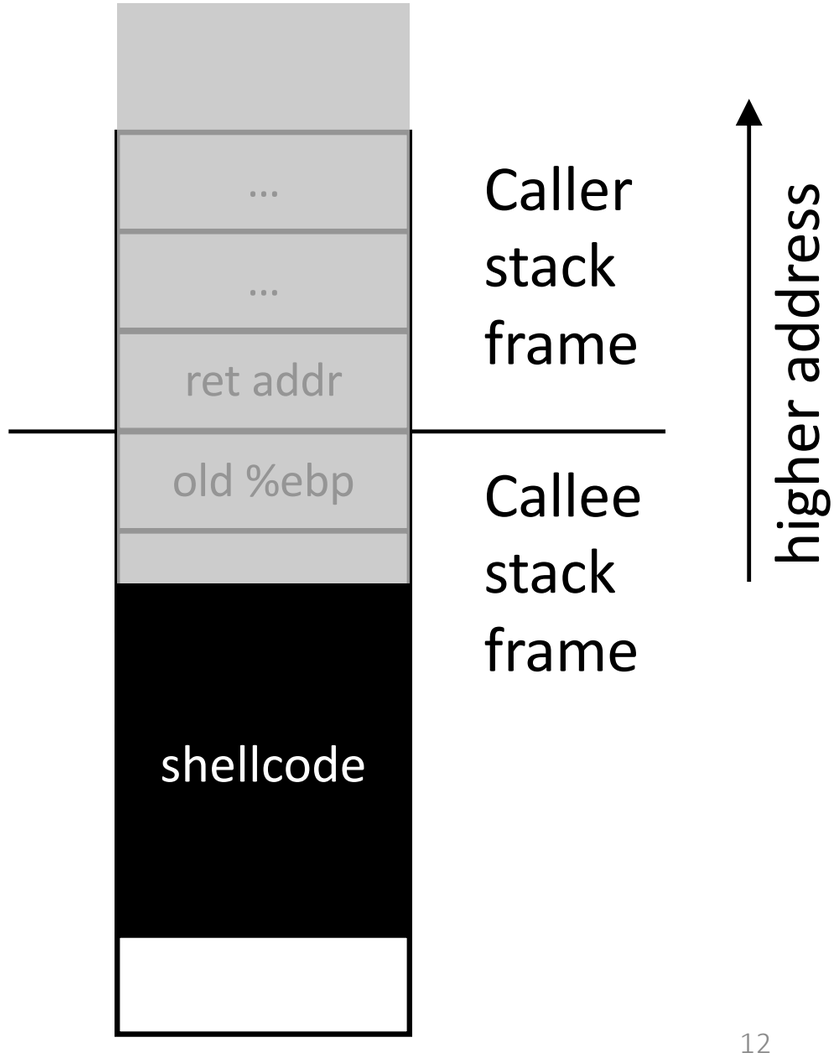
Code Injection w/ An Executable Stack 😈

- Stack buffer overflow vulnerability
  - Attackers can insert any content on stack



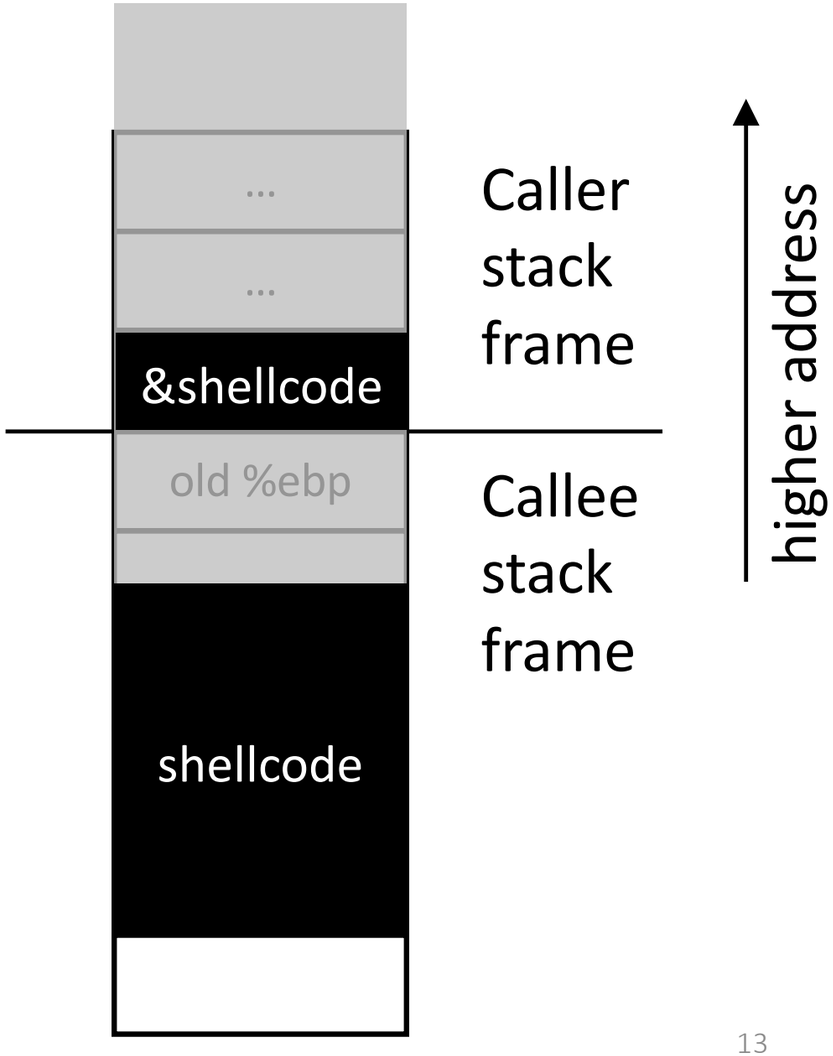
Code Injection w/ An Executable Stack 😈

- Stack buffer overflow vulnerability
  - Attackers can insert any content on stack
  1. Place shellcode on stack



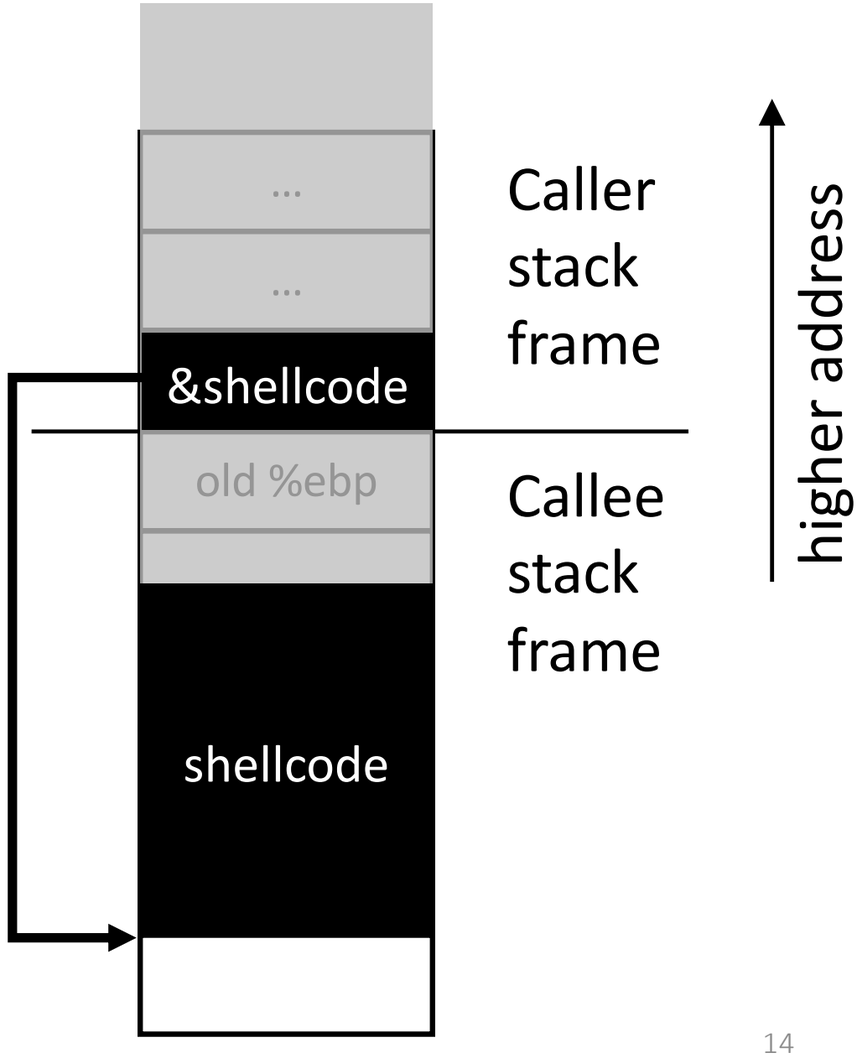
Code Injection w/ An Executable Stack 😈

- Stack buffer overflow vulnerability
  - Attackers can insert any content on stack
  1. Place shellcode on stack
  2. Overwrite return address to &shellcode



Code Injection w/ An Executable Stack 😈

- Stack buffer overflow vulnerability
  - Attackers can insert any content on stack
  1. Place shellcode on stack
  2. Overwrite return address to &shellcode
  3. Return



Stack is NOT Executable Anymore 

Stack is NOT Executable Anymore 😇

- W⊕X: writable or executable, not both

Stack is NOT Executable Anymore 😇

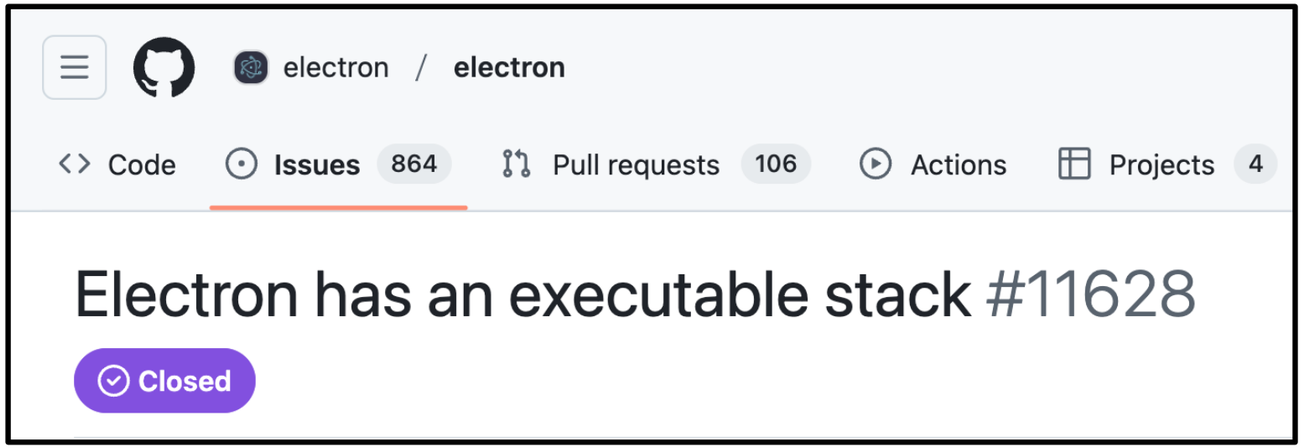
- W⊕X: writable or executable, not both
- NX bit: most significant bit of the page table entry
  - Managed & set by OS

Stack is NOT Executable Anymore 😇

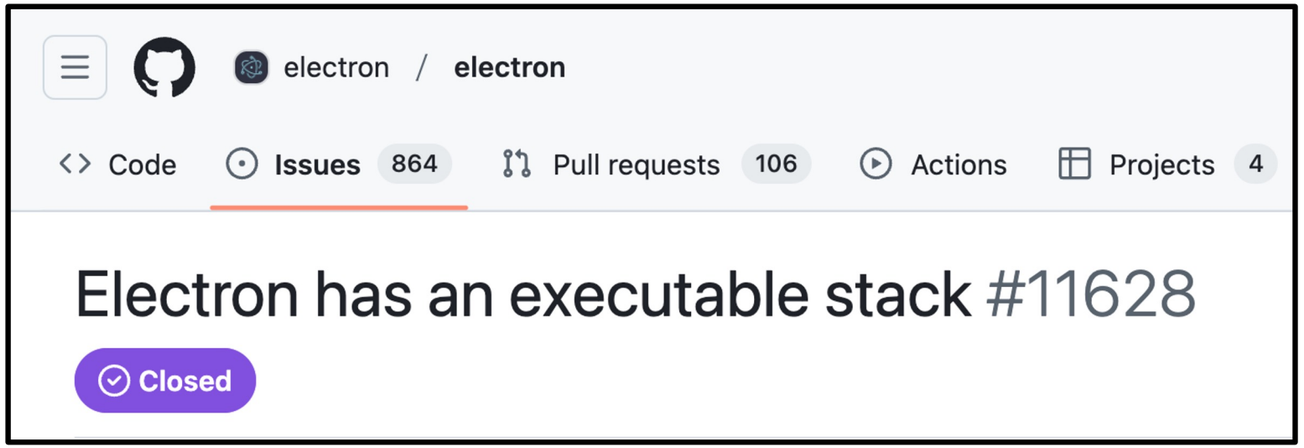
- W⊕X: writable or executable, not both
- NX bit: most significant bit of the page table entry
  - Managed & set by OS
- Widely deployed & enabled by default

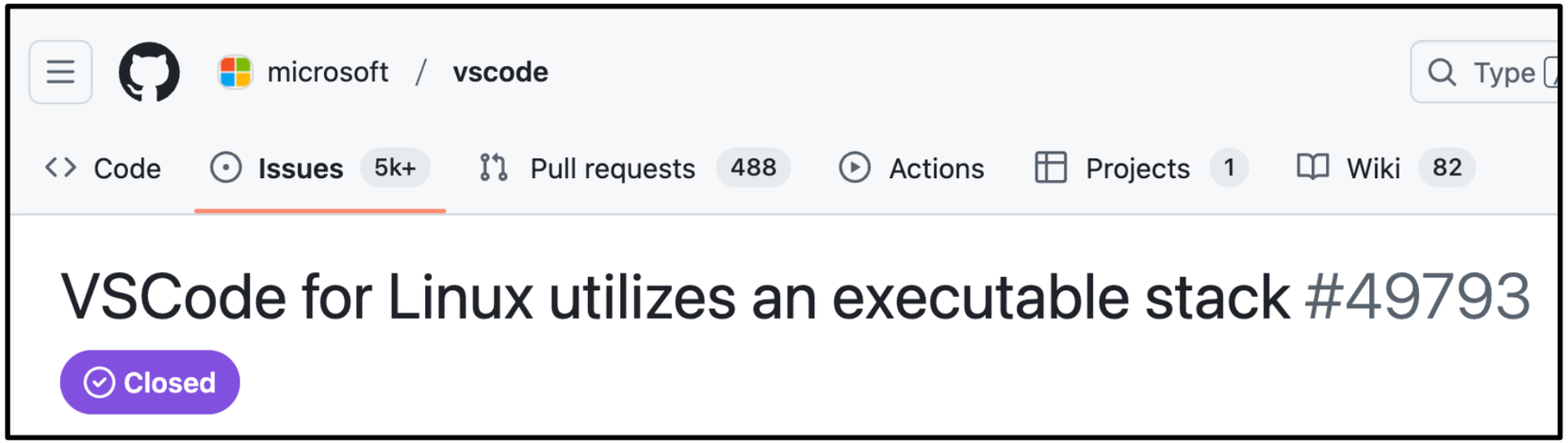
Accidentally Executable Stack

Accidentally Executable Stack

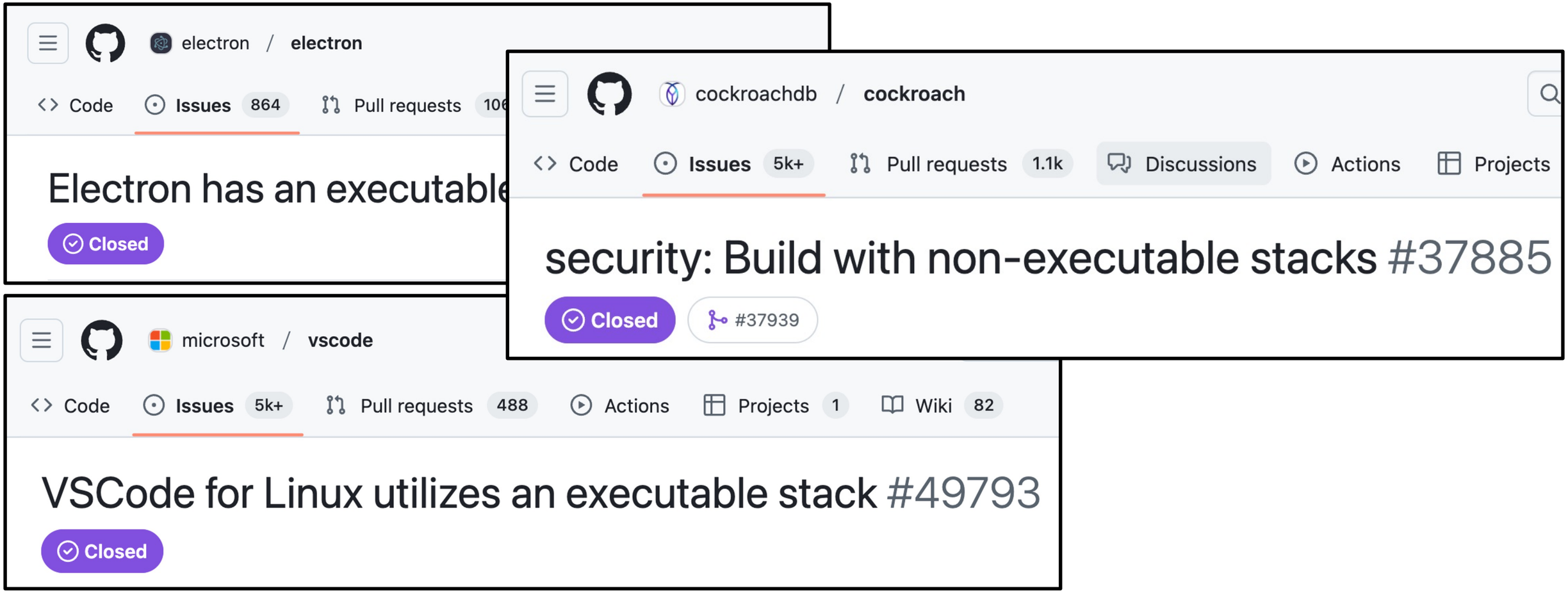


Accidentally Executable Stack

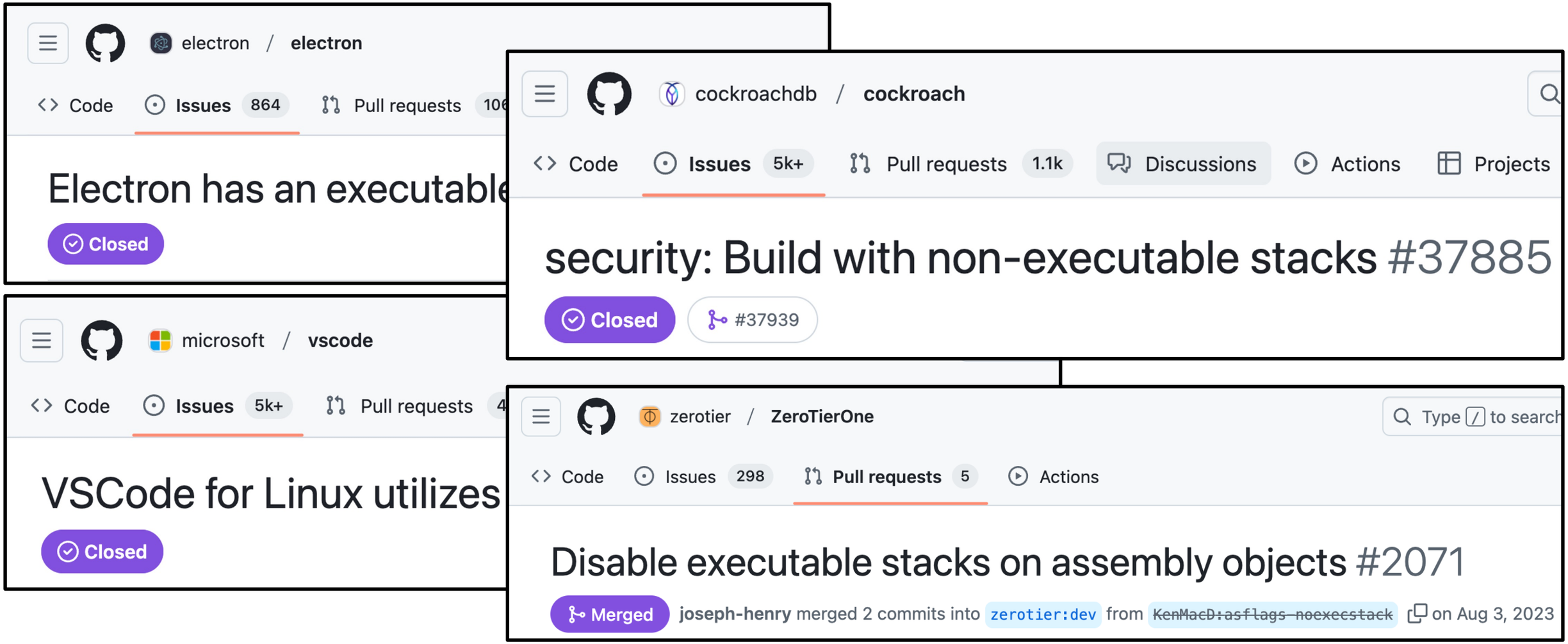




Accidentally Executable Stack



Accidentally Executable Stack



When Your Assembly is Not Good Enough

When Your Assembly is Not Good Enough

- Assembly files w/o `.section .note.GNU-stack,” ”,@progbits`

When Your Assembly is Not Good Enough

- Assembly files w/o `.section .note.GNU-stack,” ”,@progbits`
  - Missing `.note.GNU-stack` section

When Your Assembly is Not Good Enough

- Assembly files w/o `.section .note.GNU-stack,” ”,@progbits`
  - Missing `.note.GNU-stack` section ⟶ Executable stack 

When Your Assembly is Not Good Enough

- Assembly files w/o `.section .note.GNU-stack,” ”,@progbits`
  - Missing `.note.GNU-stack` section ⟶ Executable stack 
- We term this problem *BADASS* 

```
$ cat hello_world.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
        getchar();
        return 0;
}
```

```
$ cat hello_world.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
        getchar();
        return 0;
}
$ gcc hello_world.c -o hello_world
```

```
$ cat hello_world.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
        getchar();
        return 0;
}
$ gcc hello_world.c -o hello_world
$ ./hello_world
^Z[1] + Stopped                    ./hello_world
```

```
$ cat hello_world.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
        getchar();
        return 0;
}
$ gcc hello_world.c -o hello_world
$ ./hello_world
^Z[1] + Stopped                    ./hello_world
$ cat /proc/$(pgrep hello_world)/maps | grep stack
7ffc9c370000-7ffc9c391000 rw-p 00000000 00:00 0                          [stack]
```

```
$ cat hello_world.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
        getchar();
        return 0;
}
$ gcc hello_world.c -o hello_world
$ ./hello_world
^Z[1] + Stopped                    ./hello_world
$ cat /proc/$(pgrep hello_world)/maps | grep stack
7ffc9c370000-7ffc9c391000 rw-p 00000000 00:00 0                          [stack]
$ killall -9 hello_world
[1] + Killed                     ./hello_world
```

```
$ cat hello_world.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
        getchar();
        return 0;
}
$ gcc hello_world.c -o hello_world
$ ./hello_world
^Z[1] + Stopped                    ./hello_world
$ cat /proc/$(pgrep hello_world)/maps | grep stack
7ffc9c370000-7ffc9c391000 rw-p 00000000 00:00 0                          [stack]
$ killall -9 hello_world
[1] + Killed                     ./hello_world
$ cat EMPTY.s
```

```
$ cat hello_world.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
        getchar();
        return 0;
}
$ gcc hello_world.c -o hello_world
$ ./hello_world
^Z[1] + Stopped                    ./hello_world
$ cat /proc/$(pgrep hello_world)/maps | grep stack
7ffc9c370000-7ffc9c391000 rw-p 00000000 00:00 0                          [stack]
$ killall -9 hello_world
[1] + Killed                     ./hello_world
$ cat EMPTY.s
$ gcc hello_world.c EMPTY.s -o hello_world
```

```
$ cat hello_world.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
        getchar();
        return 0;
}
$ gcc hello_world.c -o hello_world
$ ./hello_world
^Z[1] + Stopped                    ./hello_world
$ cat /proc/$(pgrep hello_world)/maps | grep stack
7ffc9c370000-7ffc9c391000 rw-p 00000000 00:00 0                          [stack]
$ killall -9 hello_world
[1] + Killed                     ./hello_world
$ cat EMPTY.s
$ gcc hello_world.c EMPTY.s -o hello_world
$ ./hello_world
^Z[1] + Stopped                    ./hello_world
```

```
$ cat hello_world.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
        getchar();
        return 0;
}
$ gcc hello_world.c -o hello_world
$ ./hello_world
^Z[1] + Stopped                    ./hello_world
$ cat /proc/$(pgrep hello_world)/maps | grep stack
7ffc9c370000-7ffc9c391000 rw-p 00000000 00:00 0                          [stack]
$ killall -9 hello_world
[1] + Killed                     ./hello_world
$ cat EMPTY.s
$ gcc hello_world.c EMPTY.s -o hello_world
$ ./hello_world
^Z[1] + Stopped                    ./hello_world
$ cat /proc/$(pgrep hello_world)/maps | grep stack
7fff7cca0000-7fff7ccc1000 rwxp 00000000 00:00 0                          [stack]
```

What Happened

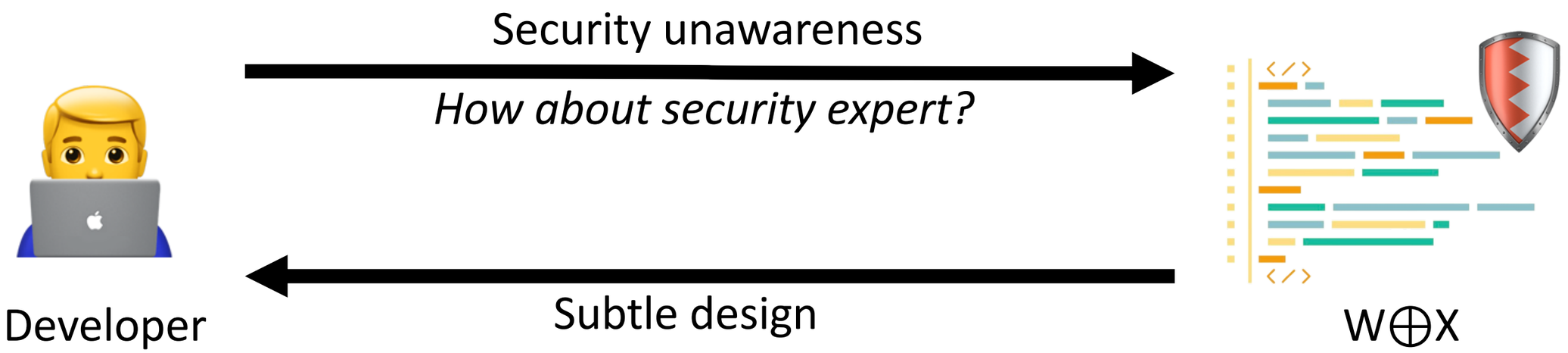
What Happened

Developer

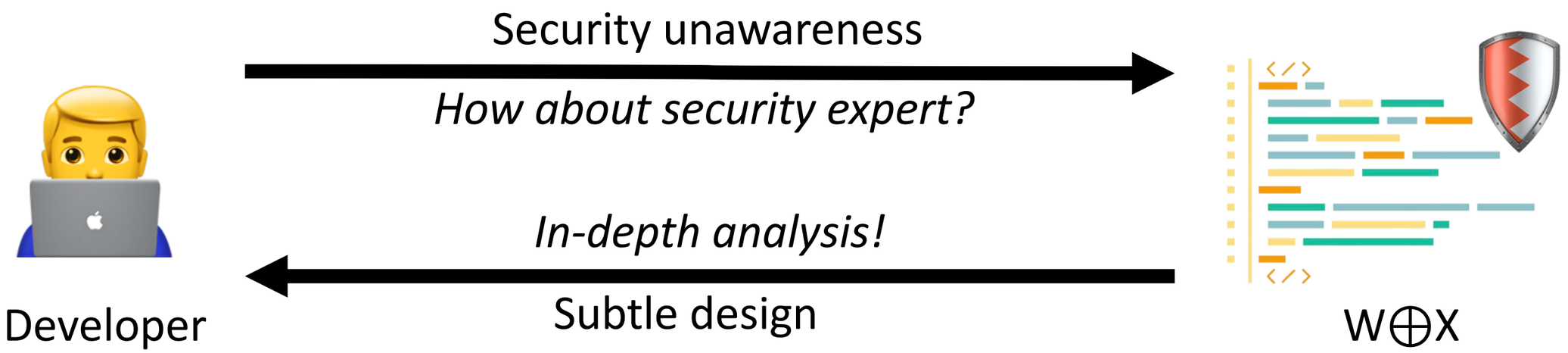
What Happened

What Happened

What Happened



What Happened



Investigating Security Applications

Investigating Security Applications

Ideal Target	

Investigating Security Applications

Ideal Target	
Utilize assembly code	

Investigating Security Applications

Ideal Target	
Utilize assembly code	
Interact with native ELF binaries	

Investigating Security Applications

Ideal Target	
Utilize assembly code	
Interact with native ELF binaries	
Open-sourced & work on Linux system	

Investigating Security Applications

Ideal Target	⭐ Inlined Reference Monitor ⭐
Utilize assembly code	
Interact with native ELF binaries	
Open-sourced & work on Linux system	

Investigating Security Applications

Ideal Target	⭐ Inlined Reference Monitor ⭐
Utilize assembly code	Assembly code for security instrumentation
Interact with native ELF binaries	
Open-sourced & work on Linux system	

Investigating Security Applications

Ideal Target	⭐ Inlined Reference Monitor ⭐
Utilize assembly code	Assembly code for security instrumentation
Interact with native ELF binaries	Output hardened binaries
Open-sourced & work on Linux system	

Investigating Security Applications

Ideal Target	⭐ Inlined Reference Monitor ⭐
Utilize assembly code	Assembly code for security instrumentation
Interact with native ELF binaries	Output hardened binaries
Open-sourced & work on Linux system	Widely open-sourced

Investigating Security Applications

Ideal Target	⭐ Inlined Reference Monitor ⭐
Utilize assembly code	Assembly code for security instrumentation
Interact with native ELF binaries	Output hardened binaries
Open-sourced & work on Linux system	Widely open-sourced

- If: *BADASS* during security instrumentatinon

Investigating Security Applications

Ideal Target	⭐ Inlined Reference Monitor ⭐
Utilize assembly code	Assembly code for security instrumentation
Interact with native ELF binaries	Output hardened binaries
Open-sourced & work on Linux system	Widely open-sourced

- If: *BADASS* during security instrumentatinon
- Then: executable stack for all “hardened” binaries

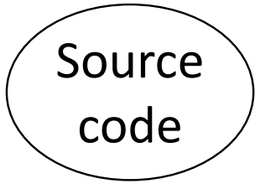
Inlined Reference Monitor (IRM)

Inlined Reference Monitor (IRM)

- Enforce security properties via injecting security checks
  - control flow integrity (CFI), software-based fault isolation (SFI)…

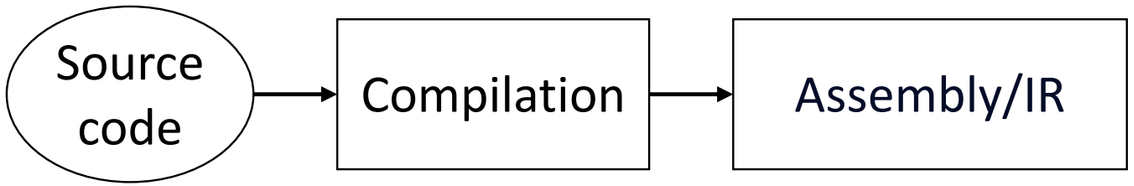
Inlined Reference Monitor (IRM)

- Enforce security properties via injecting security checks
  - control flow integrity (CFI), software-based fault isolation (SFI)…



Inlined Reference Monitor (IRM)

- Enforce security properties via injecting security checks
  - control flow integrity (CFI), software-based fault isolation (SFI)…

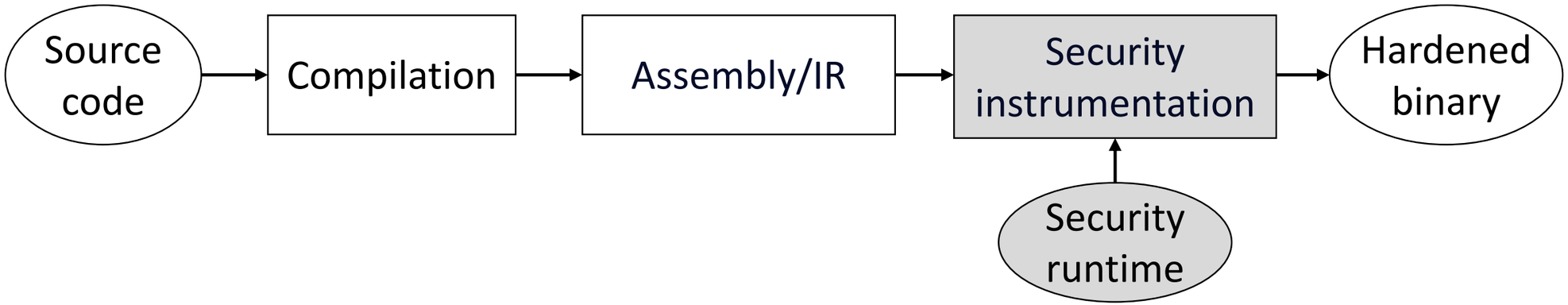


Inlined Reference Monitor (IRM)

- Enforce security properties via injecting security checks
  - control flow integrity (CFI), software-based fault isolation (SFI)…

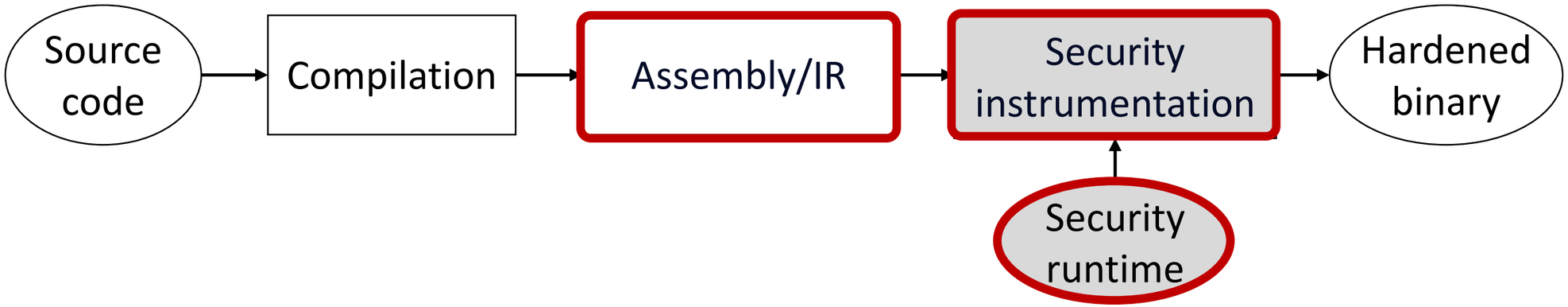
Inlined Reference Monitor (IRM)

- Enforce security properties via injecting security checks
  - control flow integrity (CFI), software-based fault isolation (SFI)…



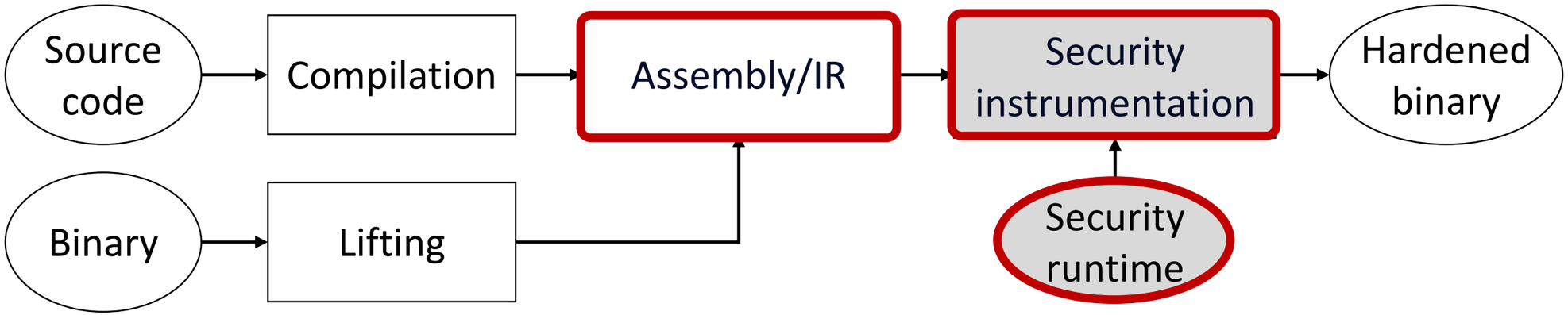
Inlined Reference Monitor (IRM)

- Enforce security properties via injecting security checks
  - control flow integrity (CFI), software-based fault isolation (SFI)…



Inlined Reference Monitor (IRM)

- Enforce security properties via injecting security checks
  - control flow integrity (CFI), software-based fault isolation (SFI)…



Investigating Security Applications

- 21 open-source IRMs
  - Nine binary rewriters
  - Eight CFI solutions
  - Four others

5/9 Binary Rewriters have *BADASS*

Binary Rewriter	Publication	Generate Assembly?	Executable Stack?	Status
Uroboros	USENIX SEC'15	yes		
Ramblr	NDSS'17	yes		
Multiverse	NDSS'18	no		
Egalito	ASPLOS'20	no		
RetroWrite	IEEE S&P'20	yes		
E9Patch	PLDI'20	no		
Ddisasm	USENIX SEC'20	yes		
ARMore	USENIX SEC'23	yes		
SAFER	USENIX SEC'23	yes		

5/9 Binary Rewriters have *BADASS*

Binary Rewriter	Publication	Generate Assembly?	Executable Stack?	Status
Uroboros	USENIX SEC'15	yes	yes	
Ramblr	NDSS'17	yes	yes	
Multiverse	NDSS'18	no	no	
Egalito	ASPLOS'20	no	no	
RetroWrite	IEEE S&P'20	yes	yes	
E9Patch	PLDI'20	no	no	
Ddisasm	USENIX SEC'20	yes	yes	
ARMore	USENIX SEC'23	yes	yes	
SAFER	USENIX SEC'23	yes	no	

5/9 Binary Rewriters have *BADASS*

Binary Rewriter	Publication	Generate Assembly?	Executable Stack?	Status
Uroboros	USENIX SEC'15	yes	yes	Checking
Ramblr	NDSS'17	yes	yes	Will fix
Multiverse	NDSS'18	no	no	-
Egalito	ASPLOS'20	no	no	-
RetroWrite	IEEE S&P'20	yes	yes	Fixed
E9Patch	PLDI'20	no	no	-
Ddisasm	USENIX SEC'20	yes	yes	Fixed (before our work)
ARMore	USENIX SEC'23	yes	yes	Fixed
SAFER	USENIX SEC'23	yes	no	-

4/8 CFI Solutions have *BADASS*

CFI	**Publication**	**Generate Assembly?**	**Executable Stack?**	**Status**
binCFI	USENIX SEC'13	yes		
MCFI	PLDI'14	yes		
LLVM CFI	USENIX SEC'14	no		
RockJIT	CCS'14	yes		
πCFI	CCS'15	yes		
PathArmor	CCS'15	yes		
μCFI	CCS'18	yes		
Android kCFI	-	no		

4/8 CFI Solutions have *BADASS*

CFI	Publication	Generate Assembly?	Executable Stack?	Status
binCFI	USENIX SEC'13	yes	no	
MCFI	PLDI'14	yes	yes	
LLVM CFI	USENIX SEC'14	no	no	
RockJIT	CCS'14	yes	yes	
πCFI	CCS'15	yes	yes	
PathArmor	CCS'15	yes	yes	
μCFI	CCS'18	yes	no	
Android kCFI	-	no	no	

4/8 CFI Solutions have *BADASS*

CFI	Publication	Generate Assembly?	Executable Stack?	Status
binCFI	USENIX SEC'13	yes	no	-
MCFI	PLDI'14	yes	yes	fixed
LLVM CFI	USENIX SEC'14	no	no	-
RockJIT	CCS'14	yes	yes	fixed
πCFI	CCS'15	yes	yes	fixed
PathArmor	CCS'15	yes	yes	won't fix
μCFI	CCS'18	yes	no	-
Android kCFI	-	no	no	-

2/2 In-process Isolation Methods have *BADASS*

Isolation	**Publication**	**Generate Assembly?**	**Executable Stack?**	**Status**
ERIM	USENIX SEC'19	yes	yes	fixed
Donky	USENIX SEC'20	yes	yes	no risk

Takeaway 1

Developer

Security unawareness

How about security expert?

W⊕X

Takeaway 1

Developer

Security unawareness

How about security expert?

W⊕X

- Even experienced security researchers/developers may miss the directive

Takeaway 1

Developer

Security unawareness

How about security expert?

W⊕X

- Even experienced security researchers/developers may miss the directive
- Cannot simply attribute *BADASS* issues to the security unawareness

Takeaway 1

Developer

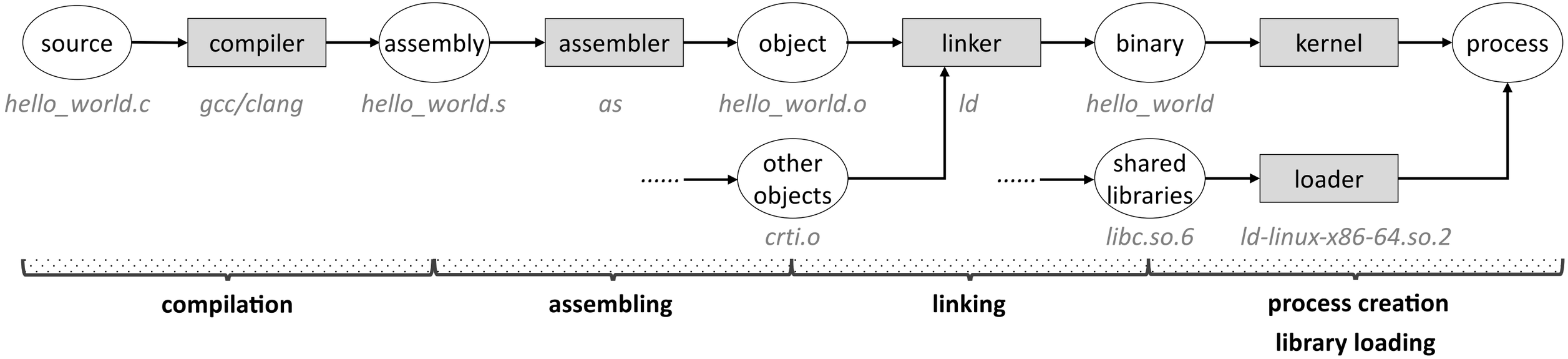
Security unawareness

How about security expert?

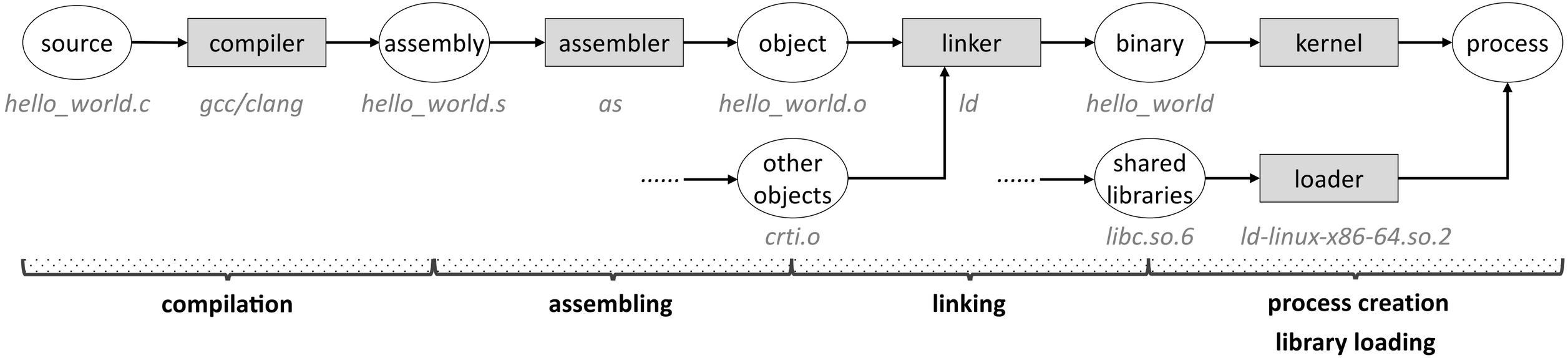
W⊕X

- Even experienced security researchers/developers may miss the directive
- Cannot simply attribute *BADASS* issues to the security unawareness
- An in-depth analysis of W⊕X enforcement is necessary

W⊕X Enforcement Analysis

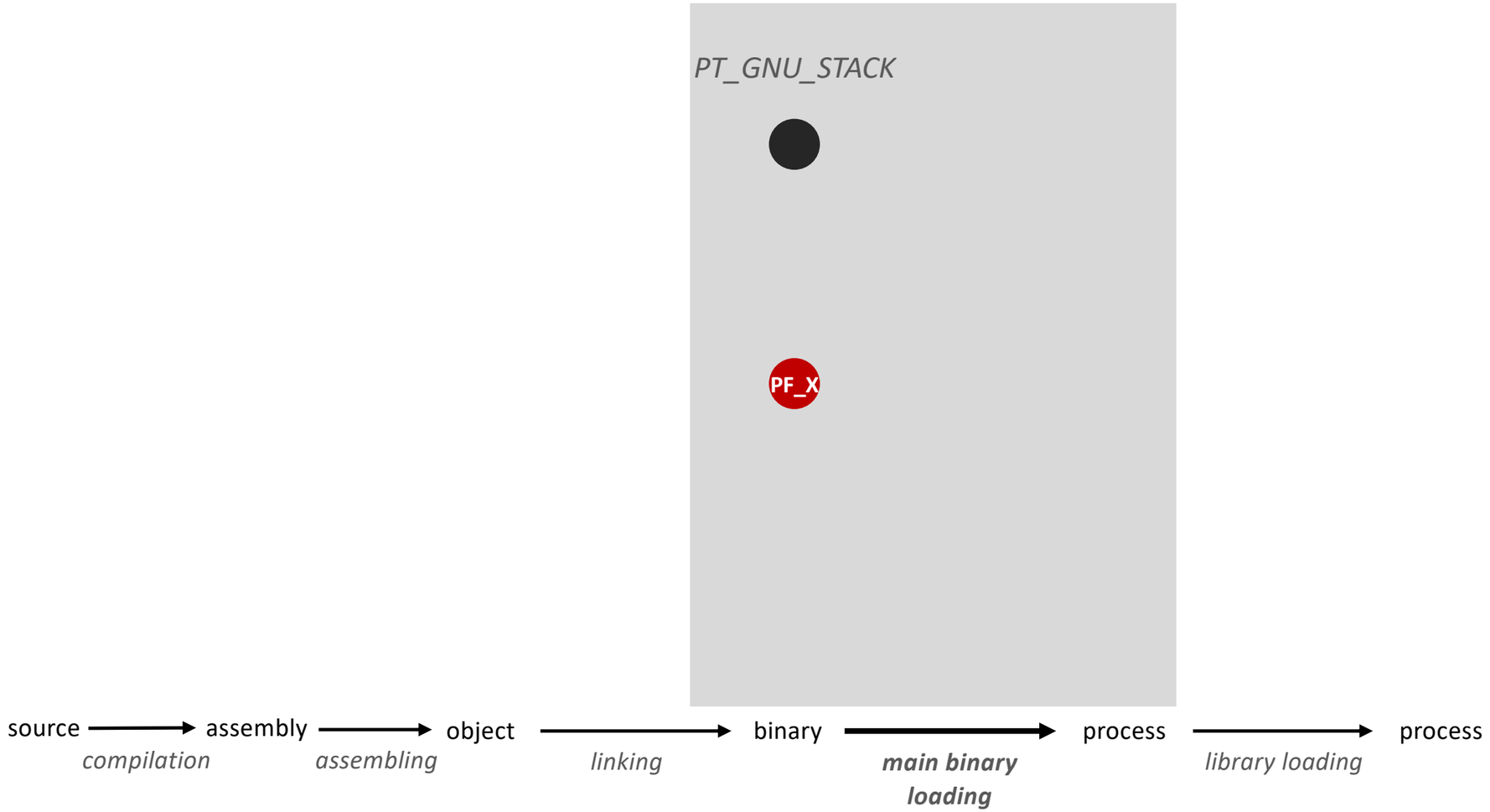


W⊕X Enforcement Analysis



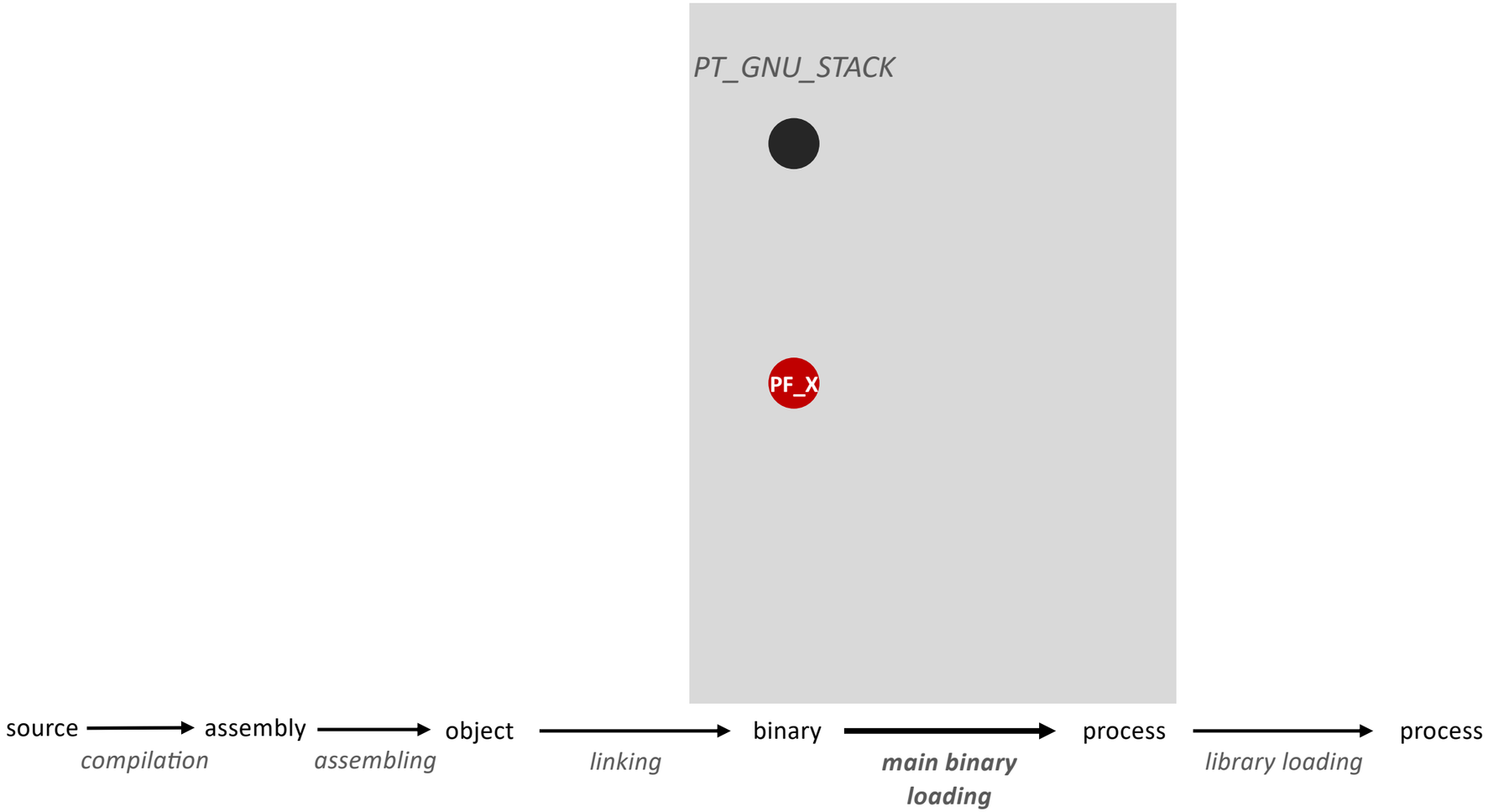
- **ALL** steps are related to stack permission

Stack Permission (Process Creation)



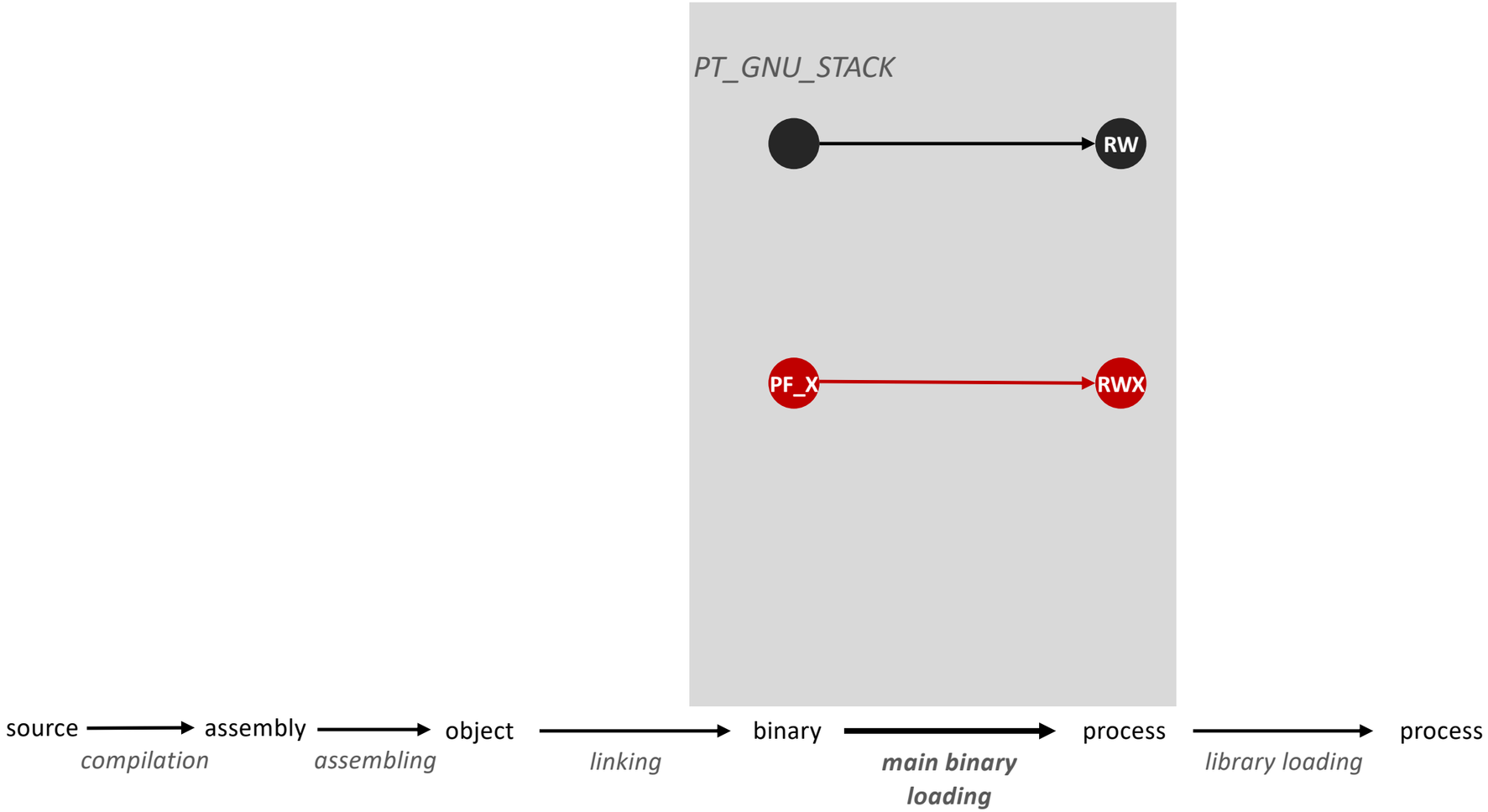
Stack Permission (Process Creation)

- Kernel checks **program header** PT_GNU_STACK of main binary

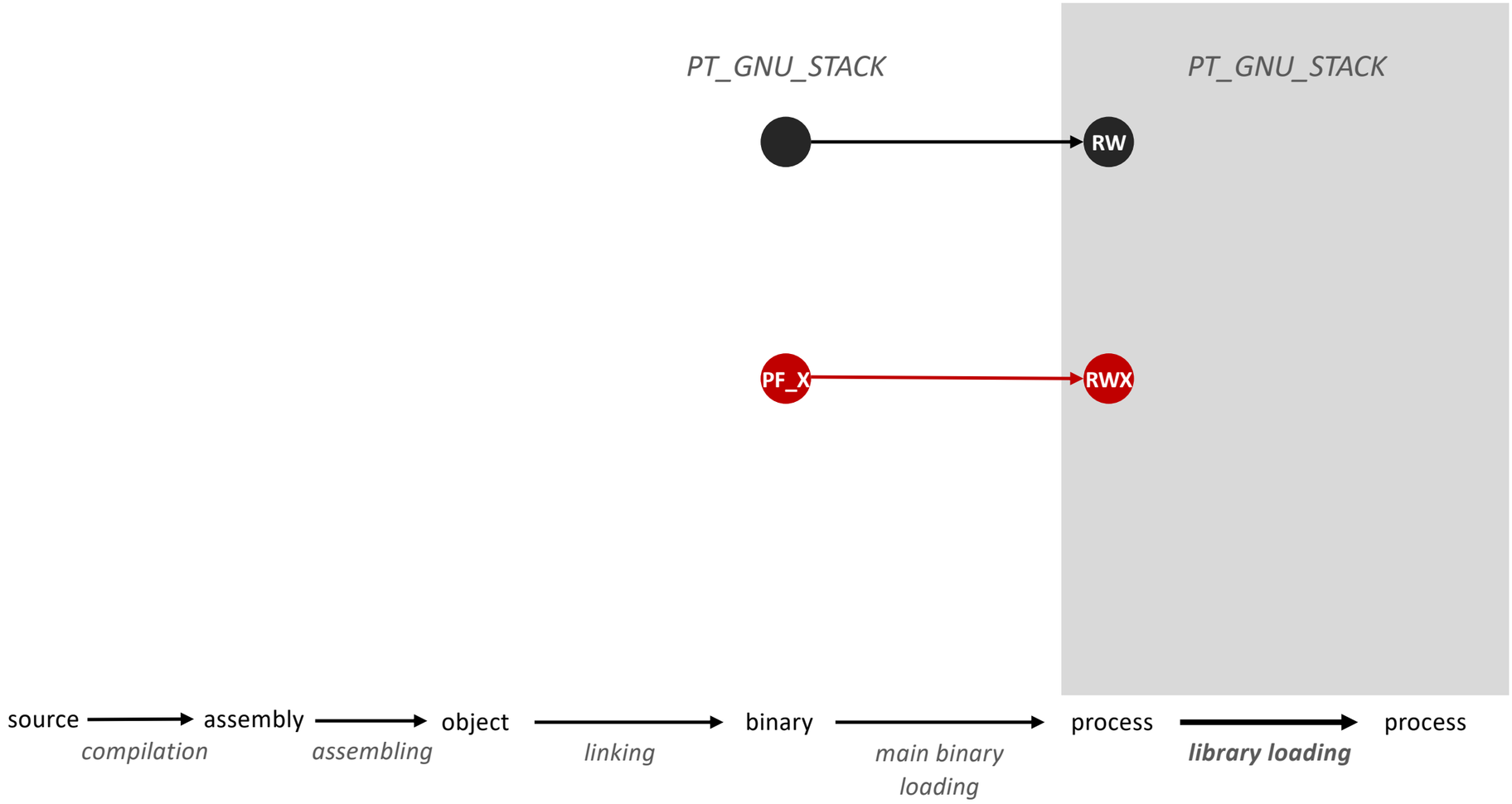


Stack Permission (Process Creation)

- Kernel checks **program header** PT_GNU_STACK of main binary

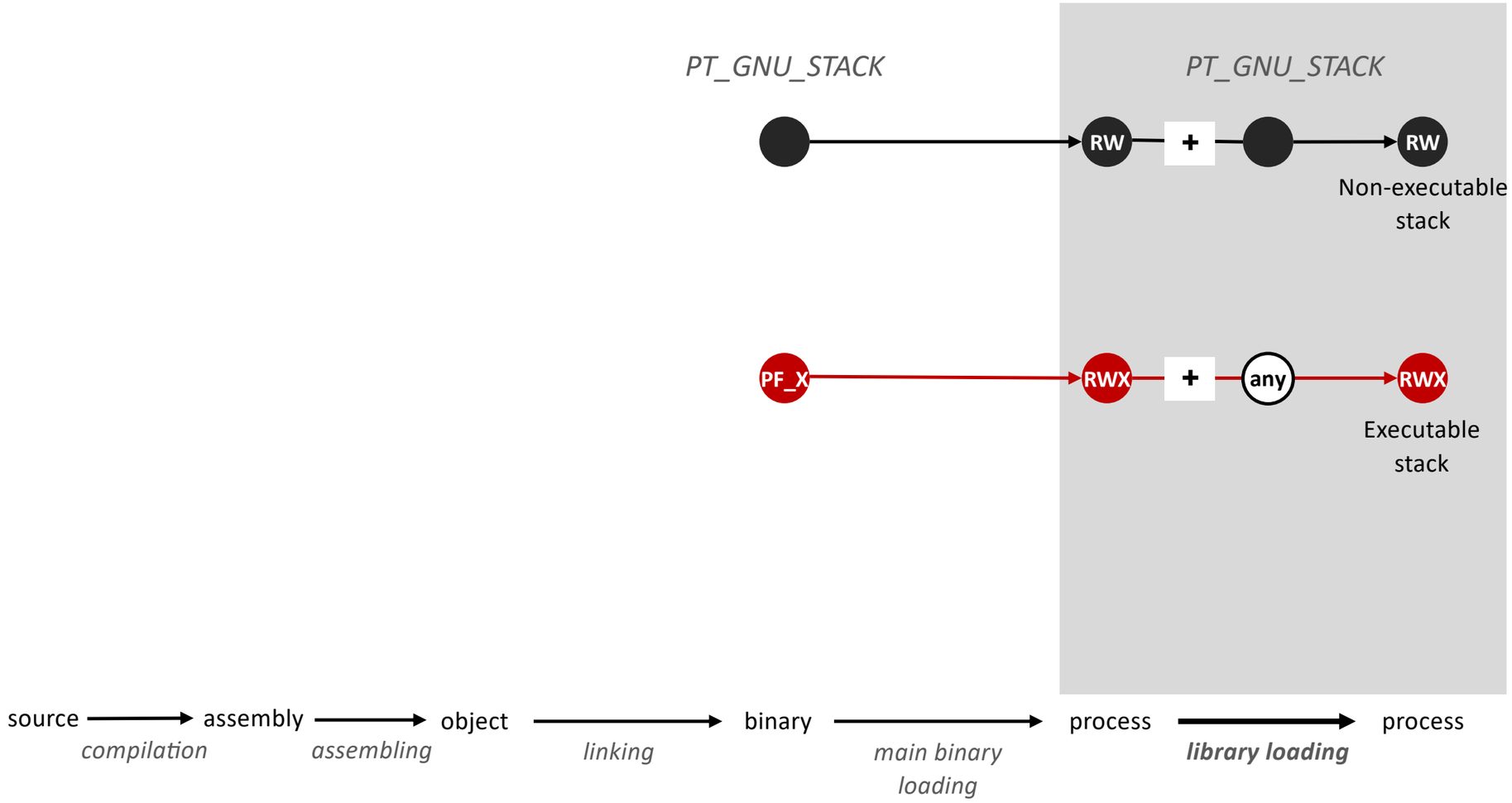


Stack Permission (Dynamic Library Loading)



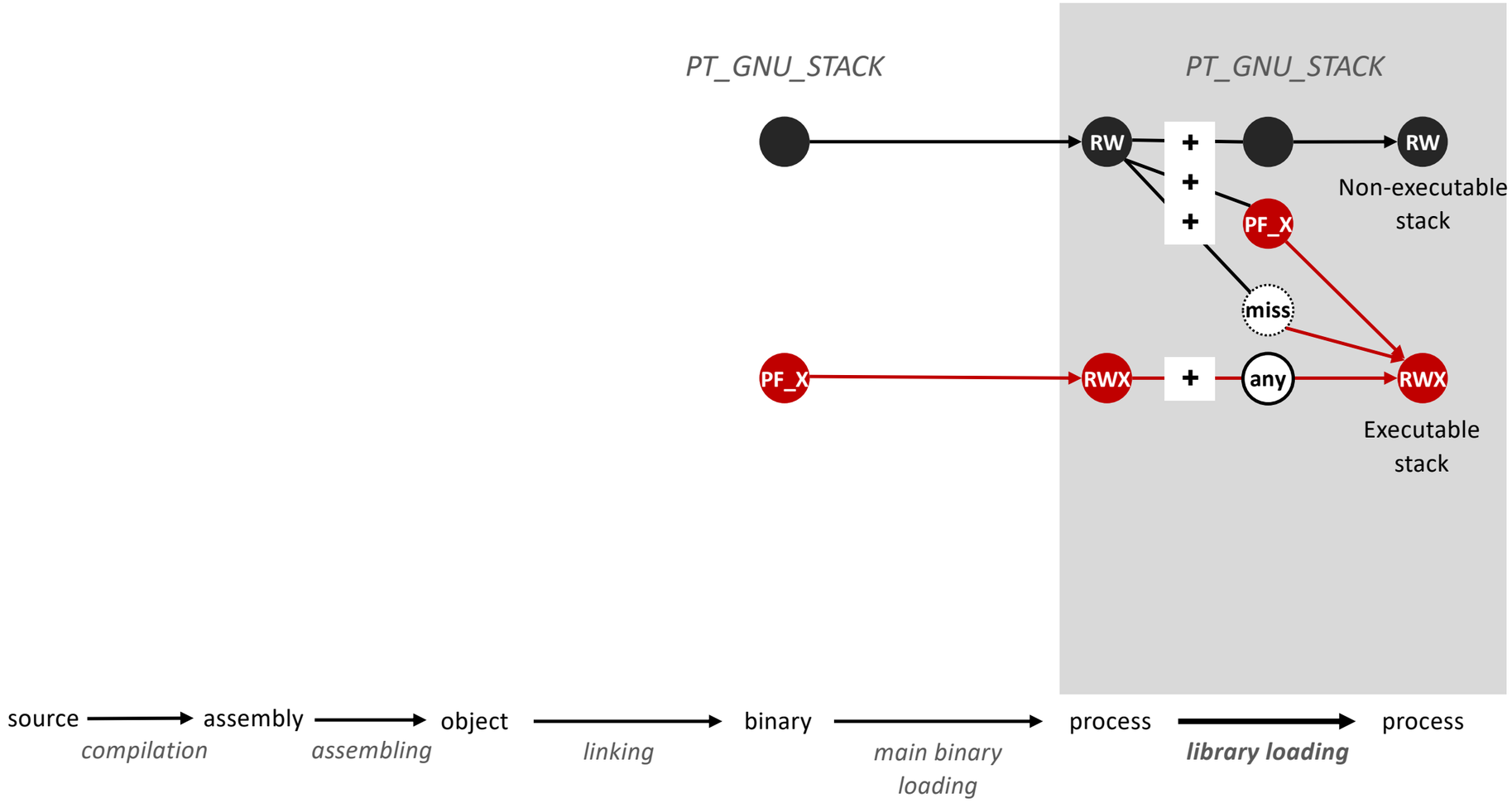
Stack Permission (Dynamic Library Loading)

- Loader checks the **program header** PT_GNU_STACK of each library

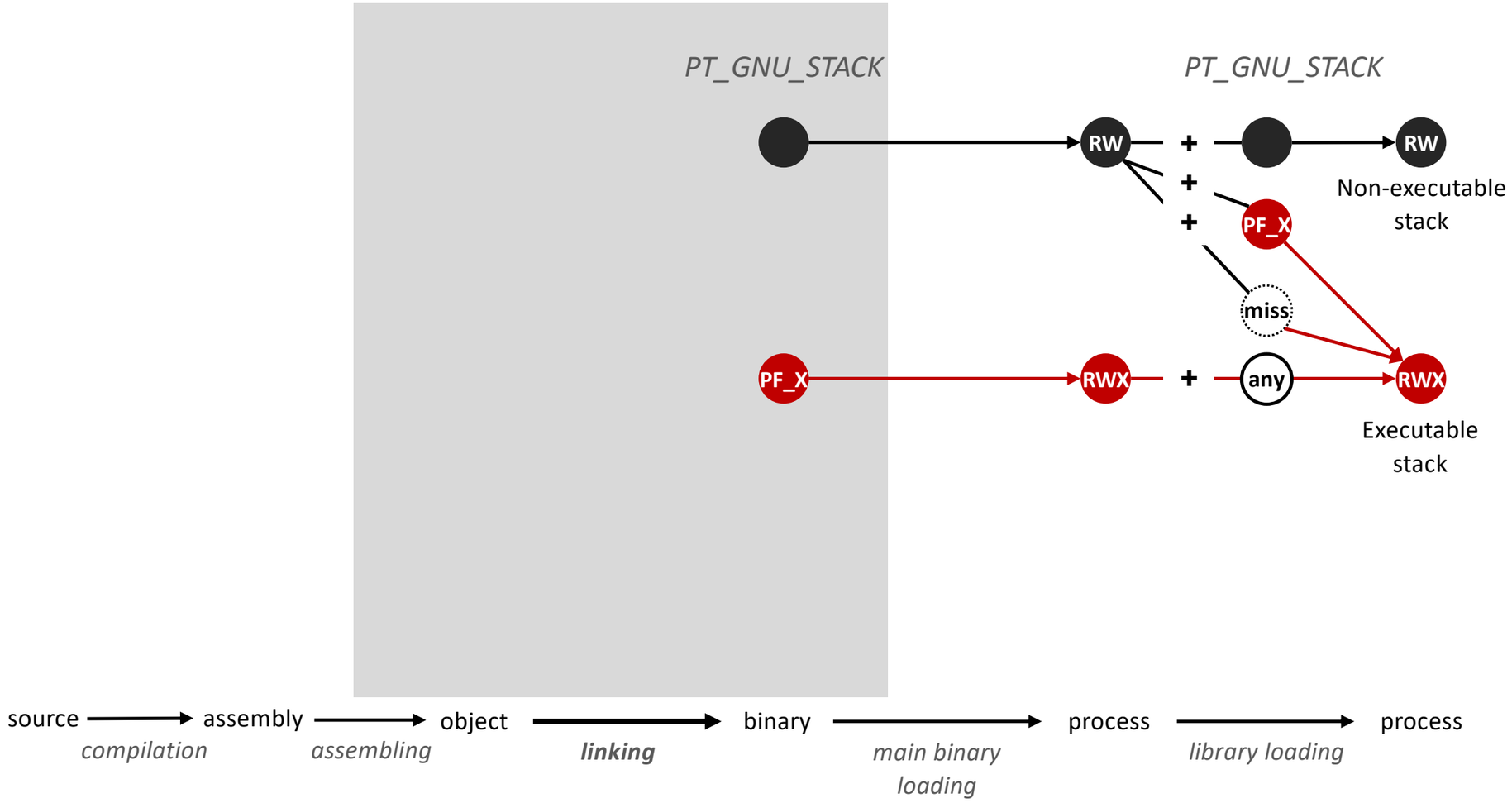


Stack Permission (Dynamic Library Loading)

- Loader checks the **program header** PT_GNU_STACK of each library

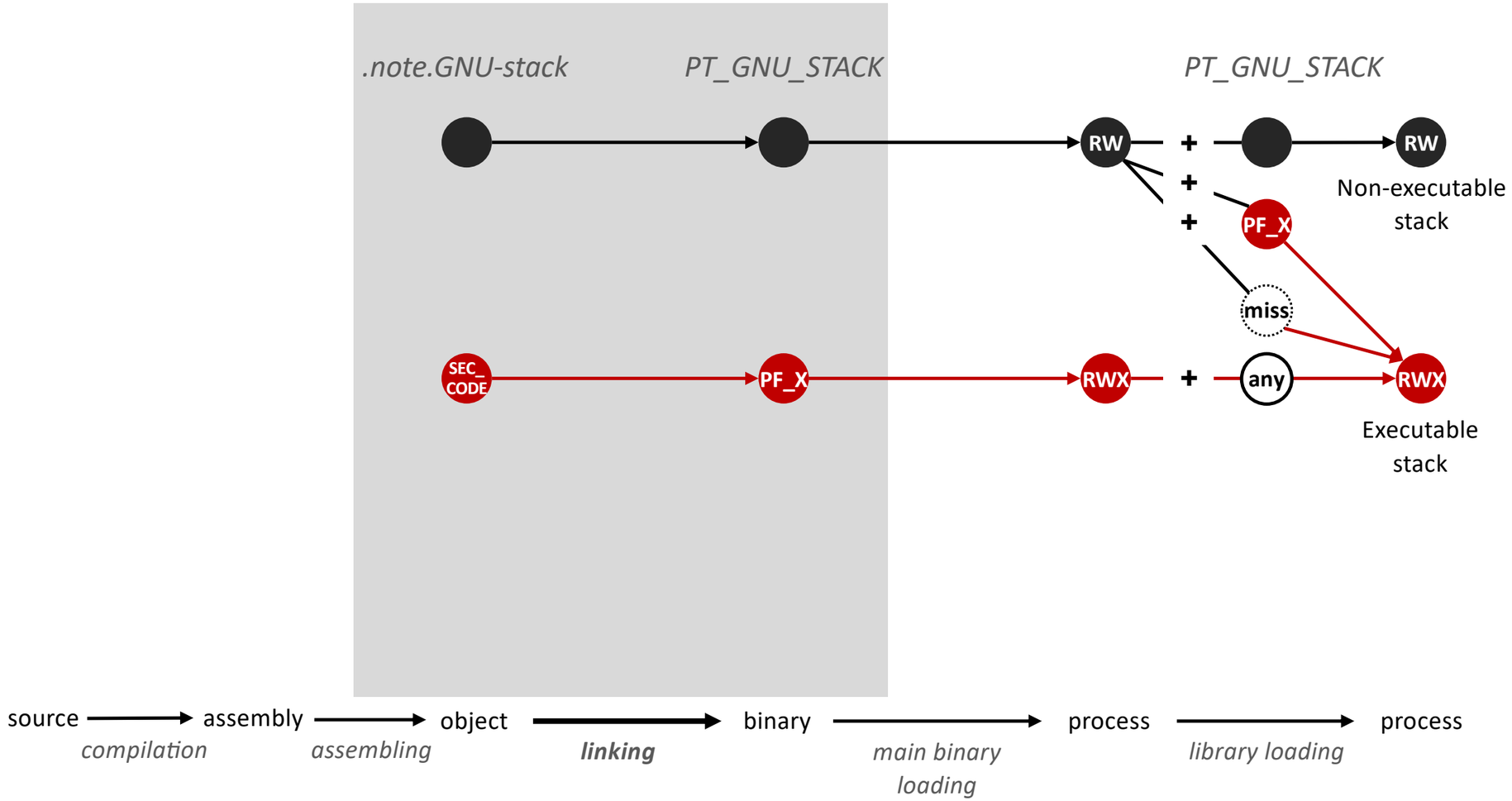


Stack Permission (Linking)



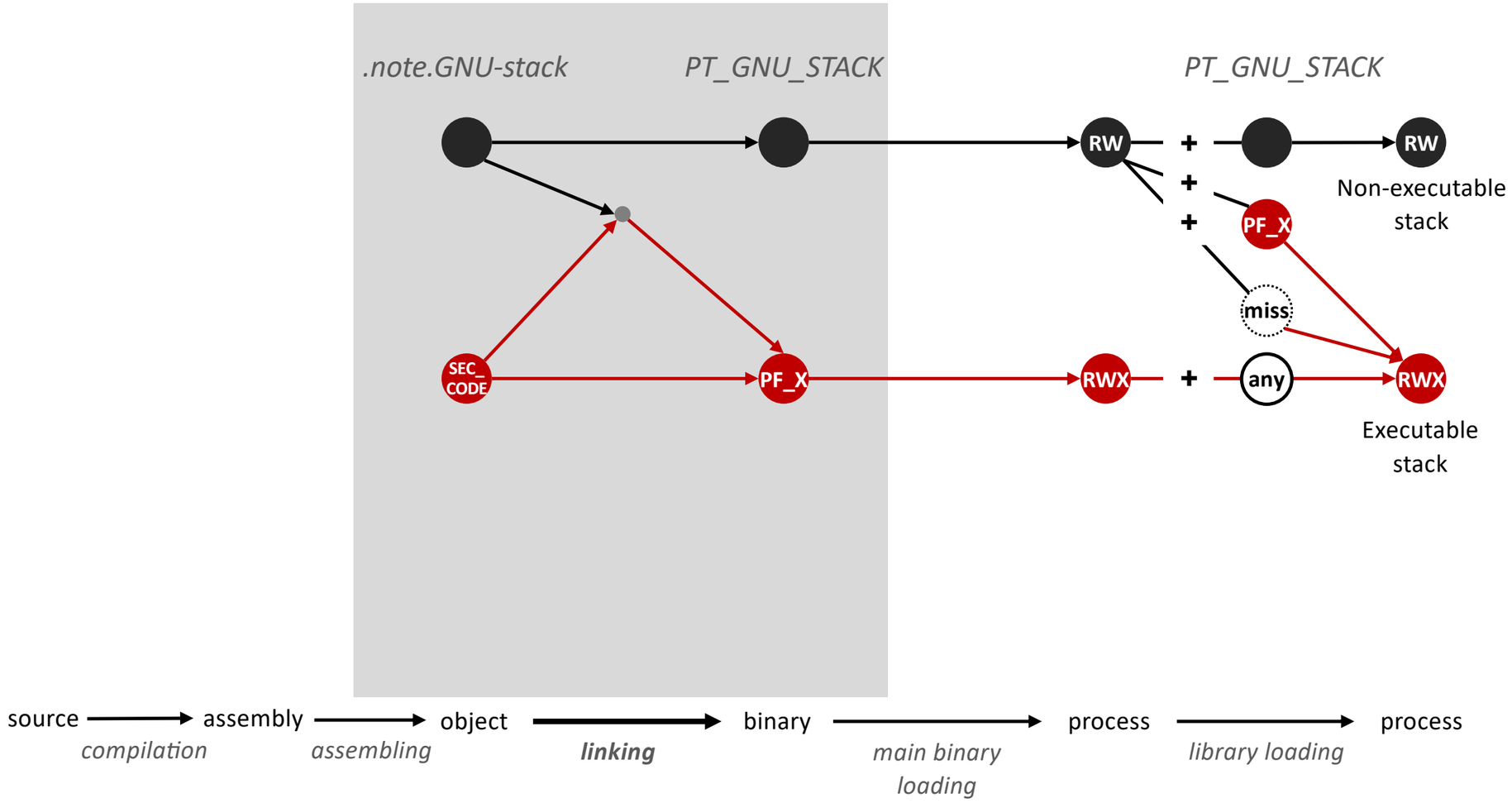
Stack Permission (Linking)

- Linker checks the **section header** .note.GNU-stack of each object file



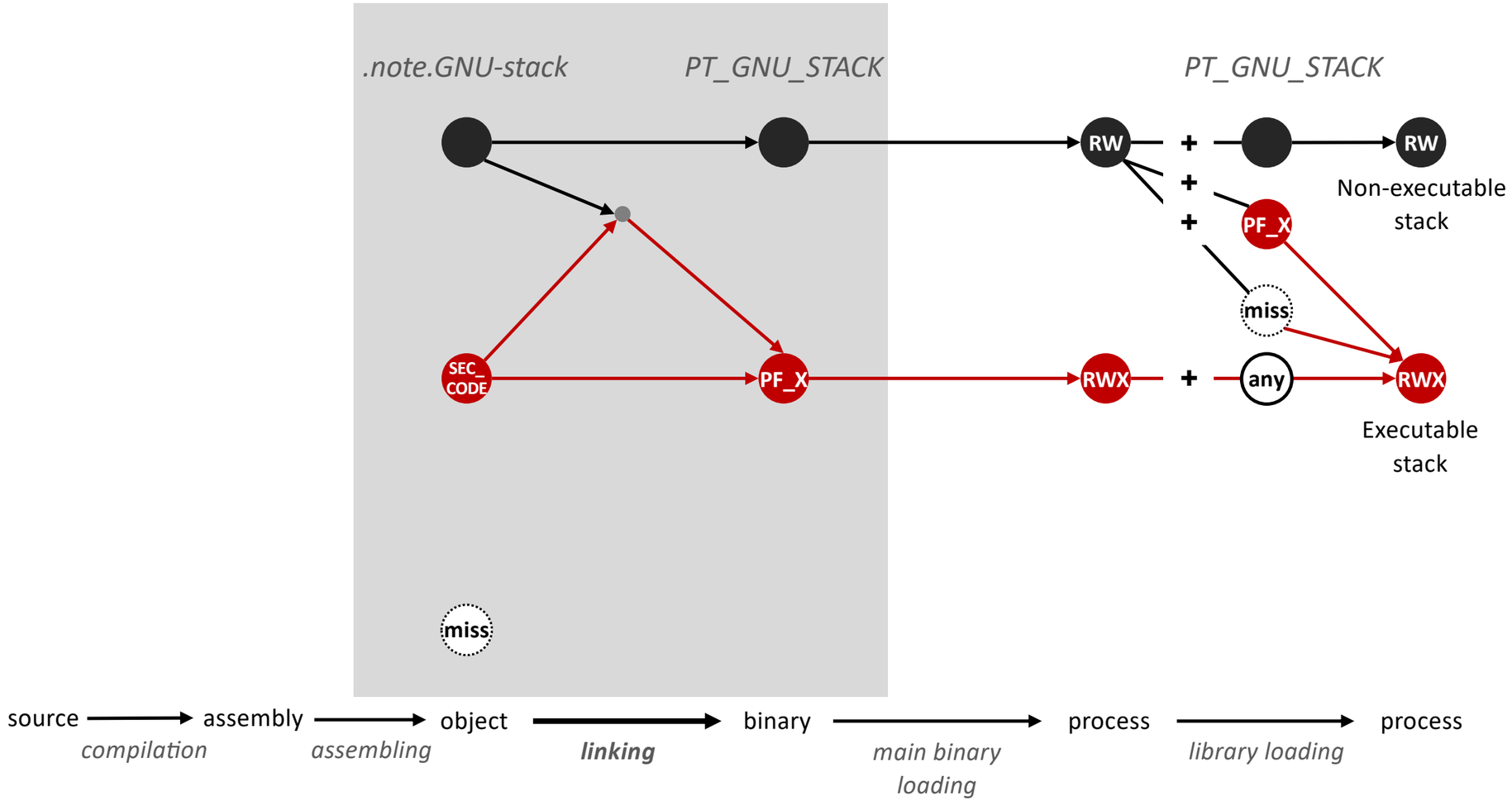
Stack Permission (Linking)

- Linker checks the **section header** .note.GNU-stack of each object file



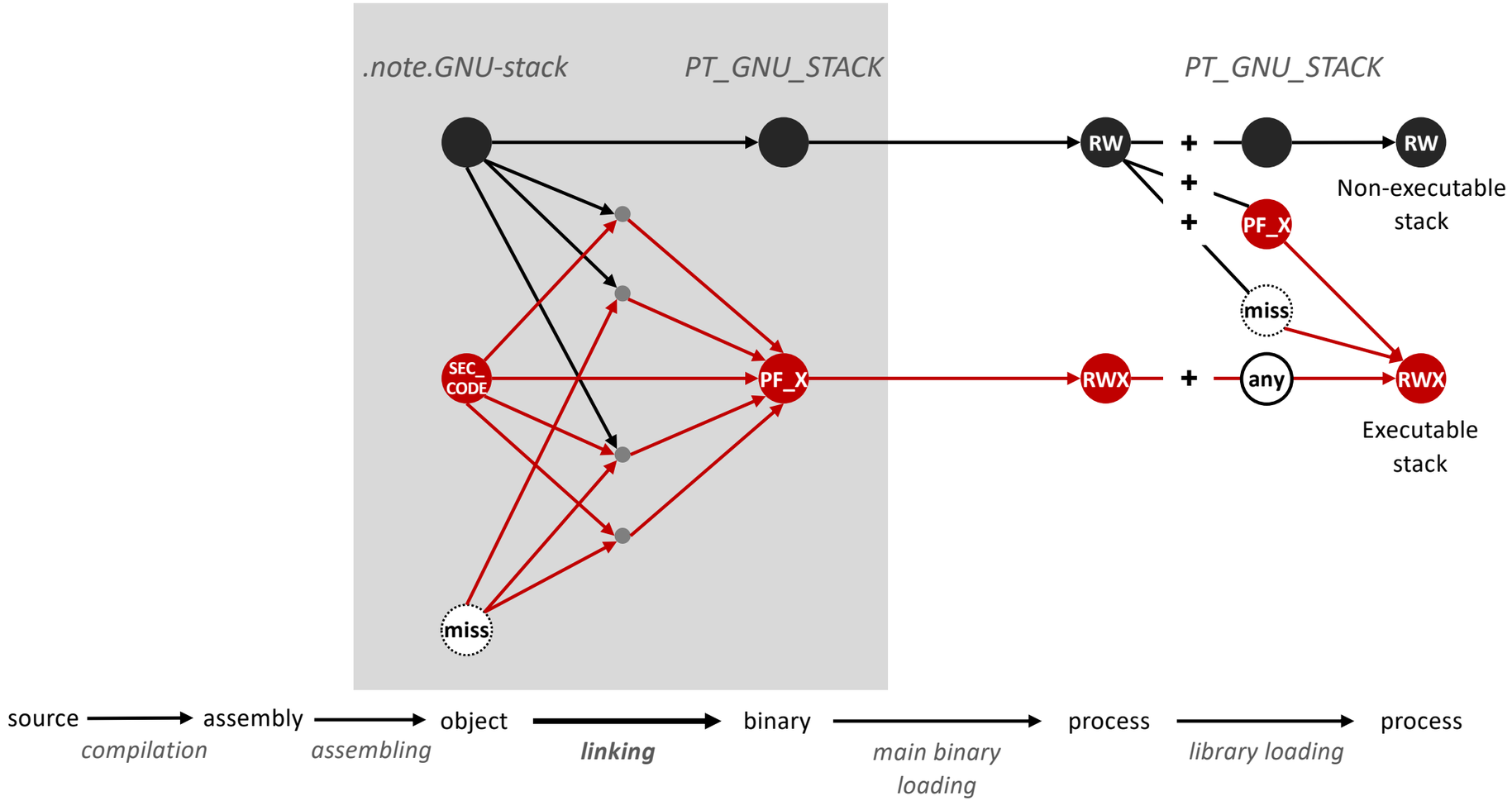
Stack Permission (Linking)

- Linker checks the **section header** .note.GNU-stack of each object file

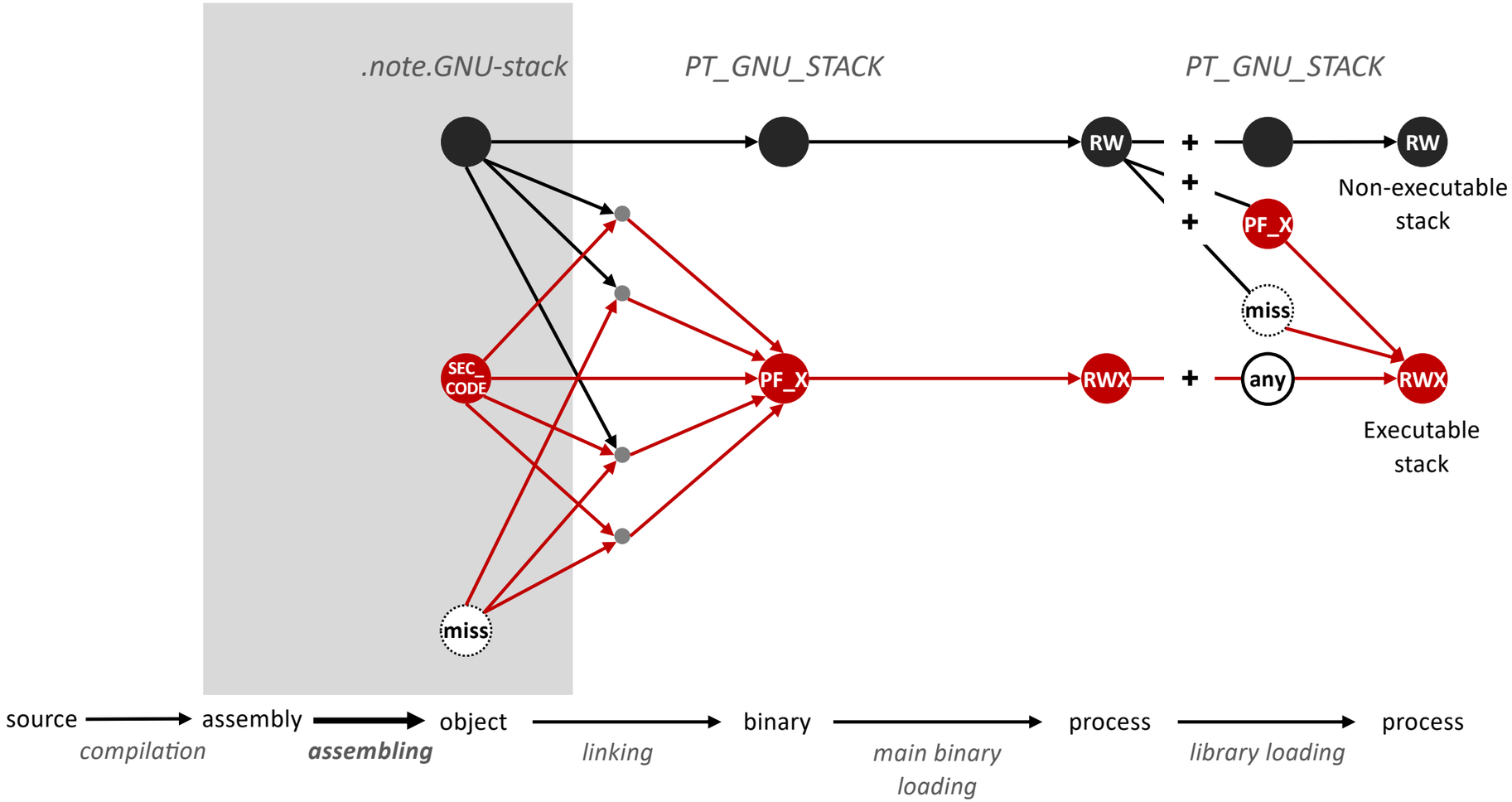


Stack Permission (Linking)

- Linker checks the **section header** .note.GNU-stack of each object file

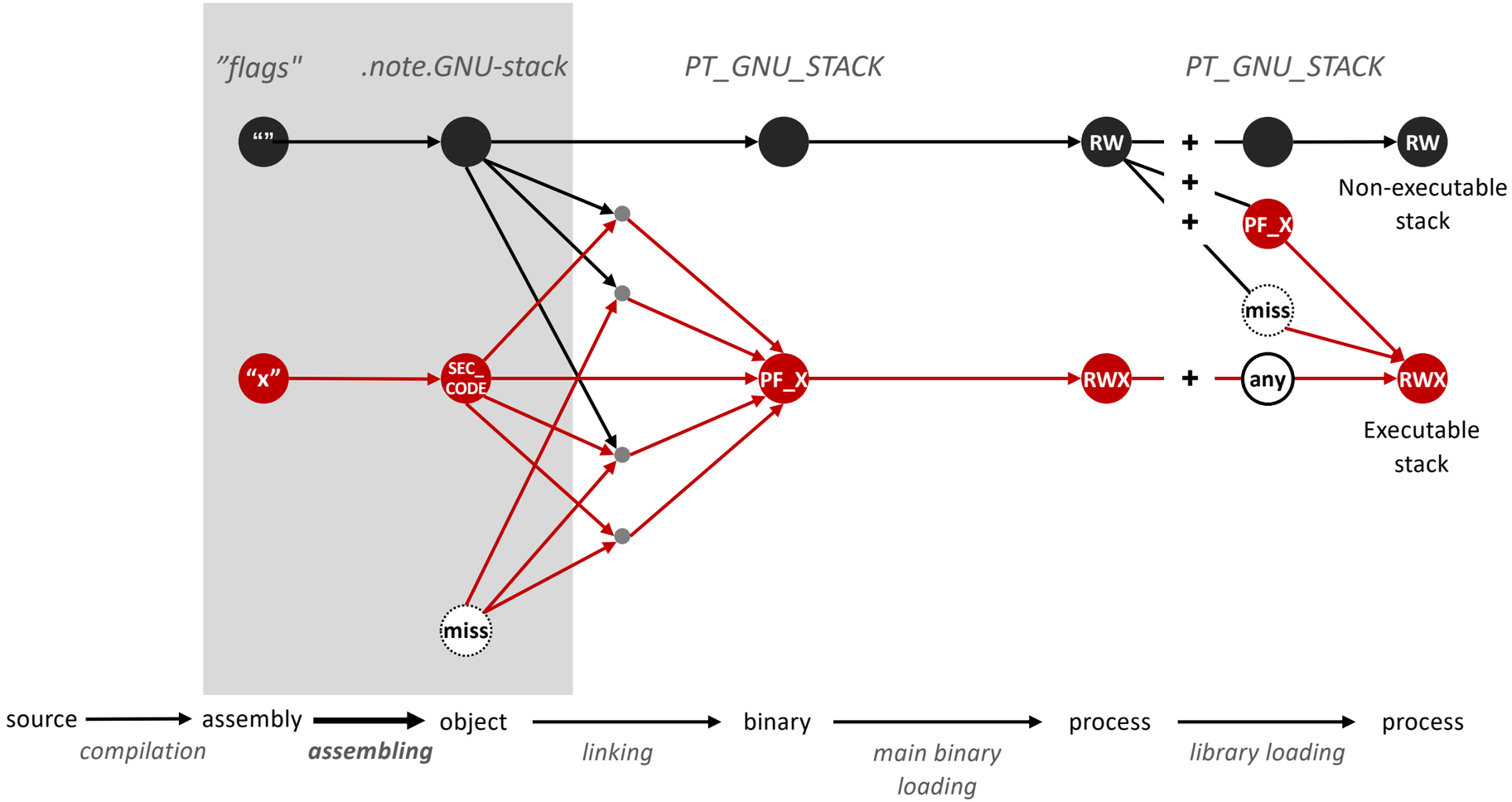


Stack Permission (Assembling)



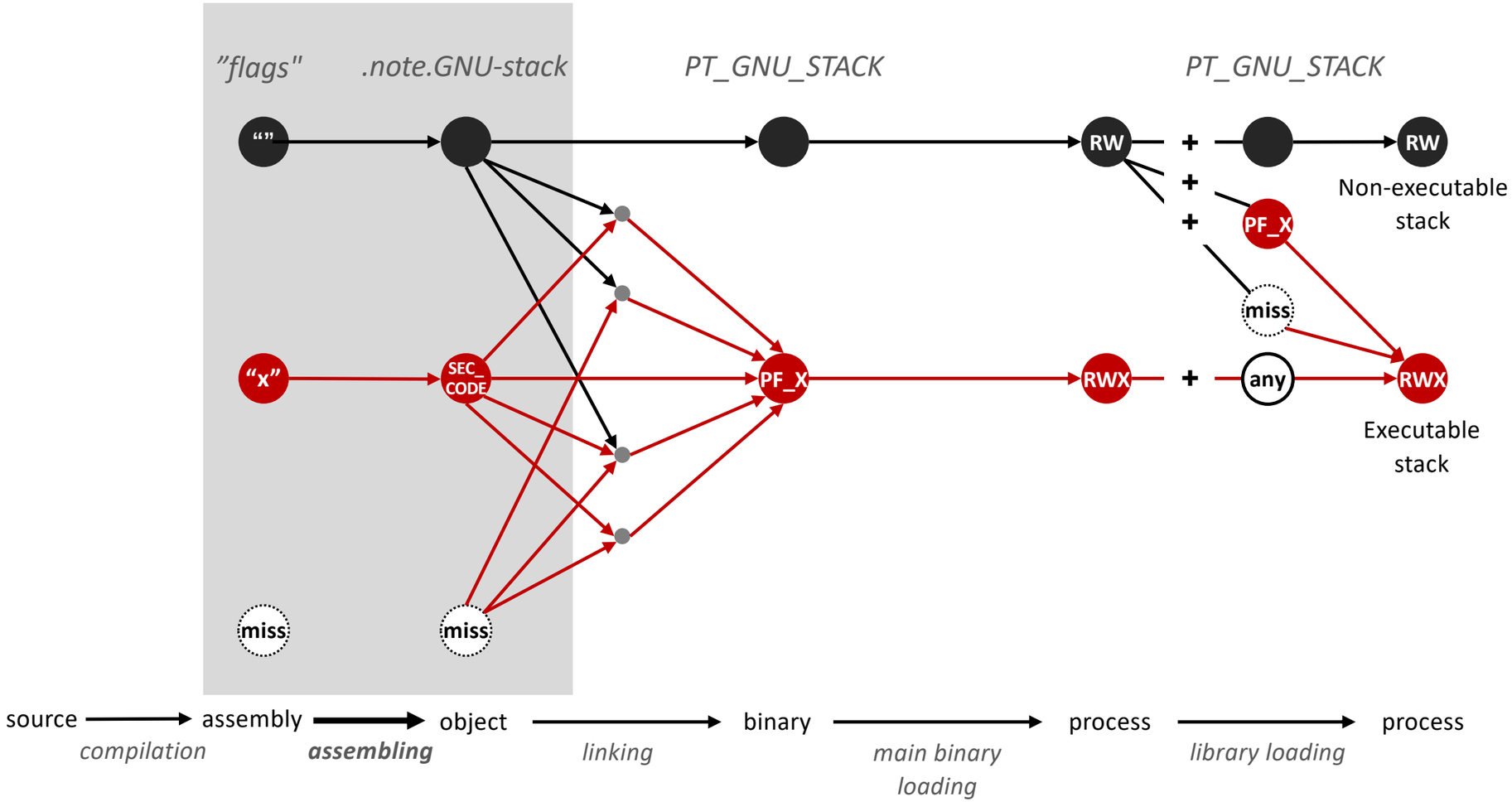
Stack Permission (Assembling)

- Assembler checks assembly file for `.section .note.GNU-stack,"flags”,@progbits`



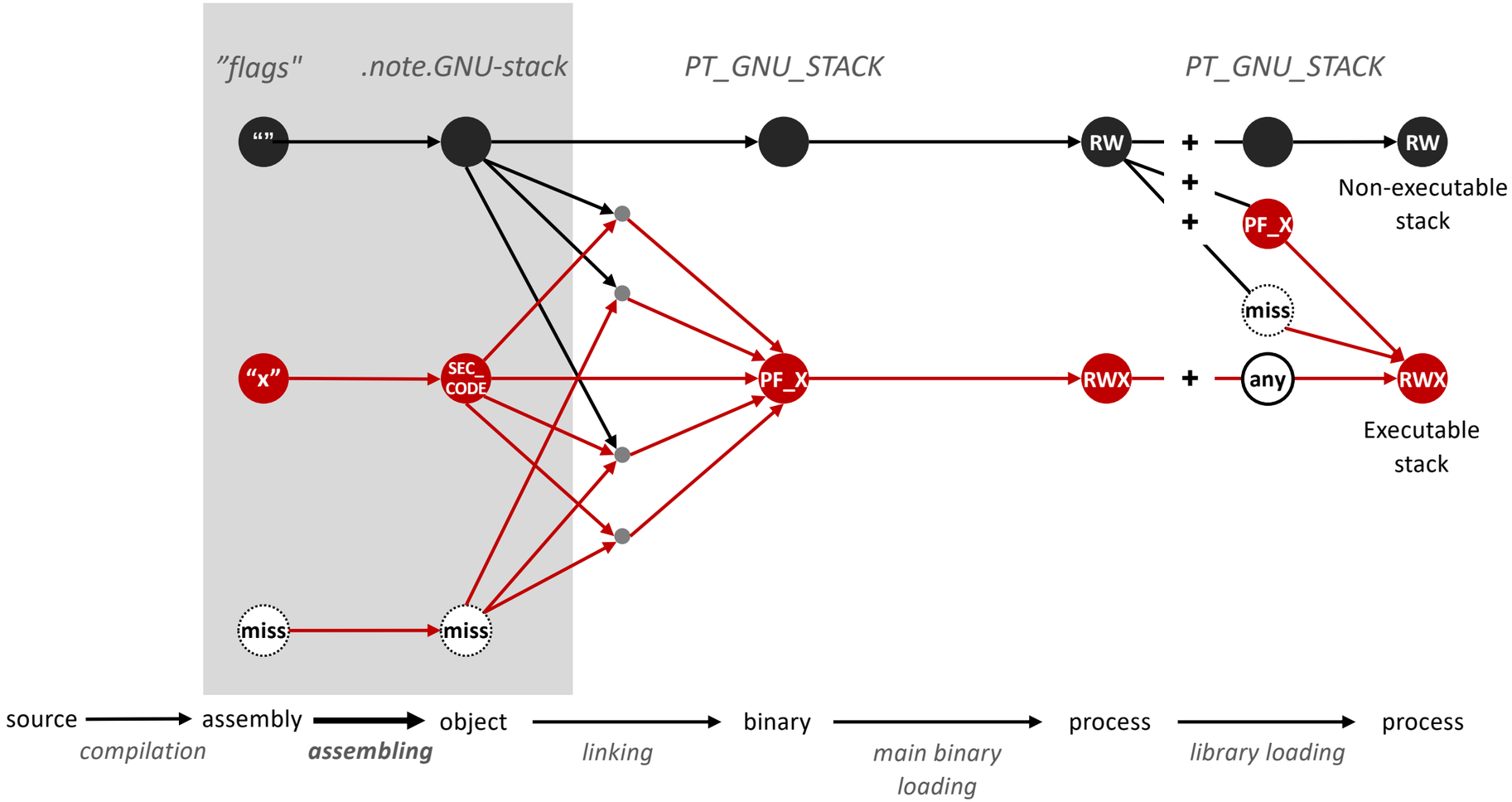
Stack Permission (Assembling)

- Assembler checks assembly file for `.section .note.GNU-stack,"flags”,@progbits`

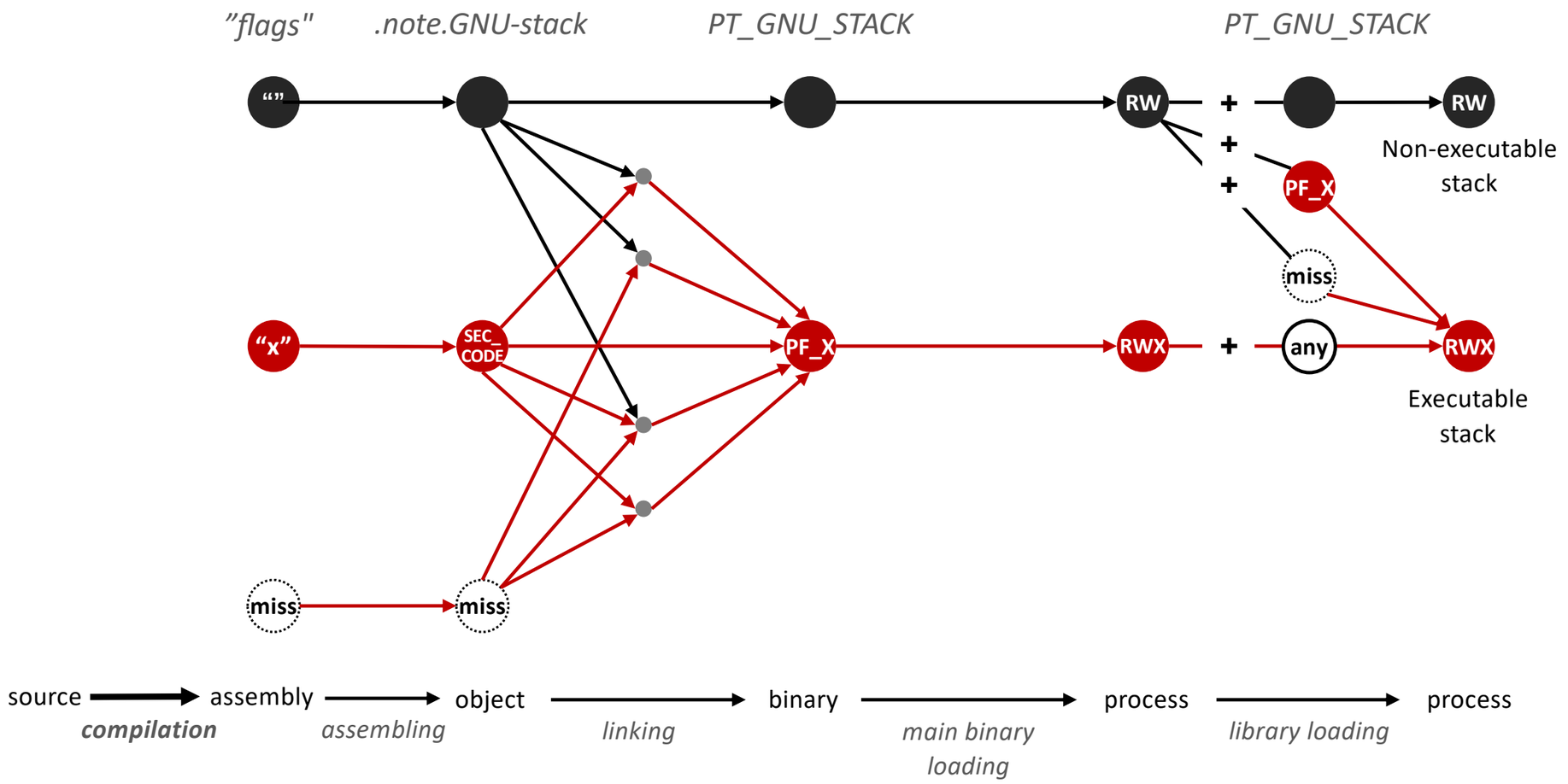


Stack Permission (Assembling)

- Assembler checks assembly file for `.section .note.GNU-stack,"flags”,@progbits`

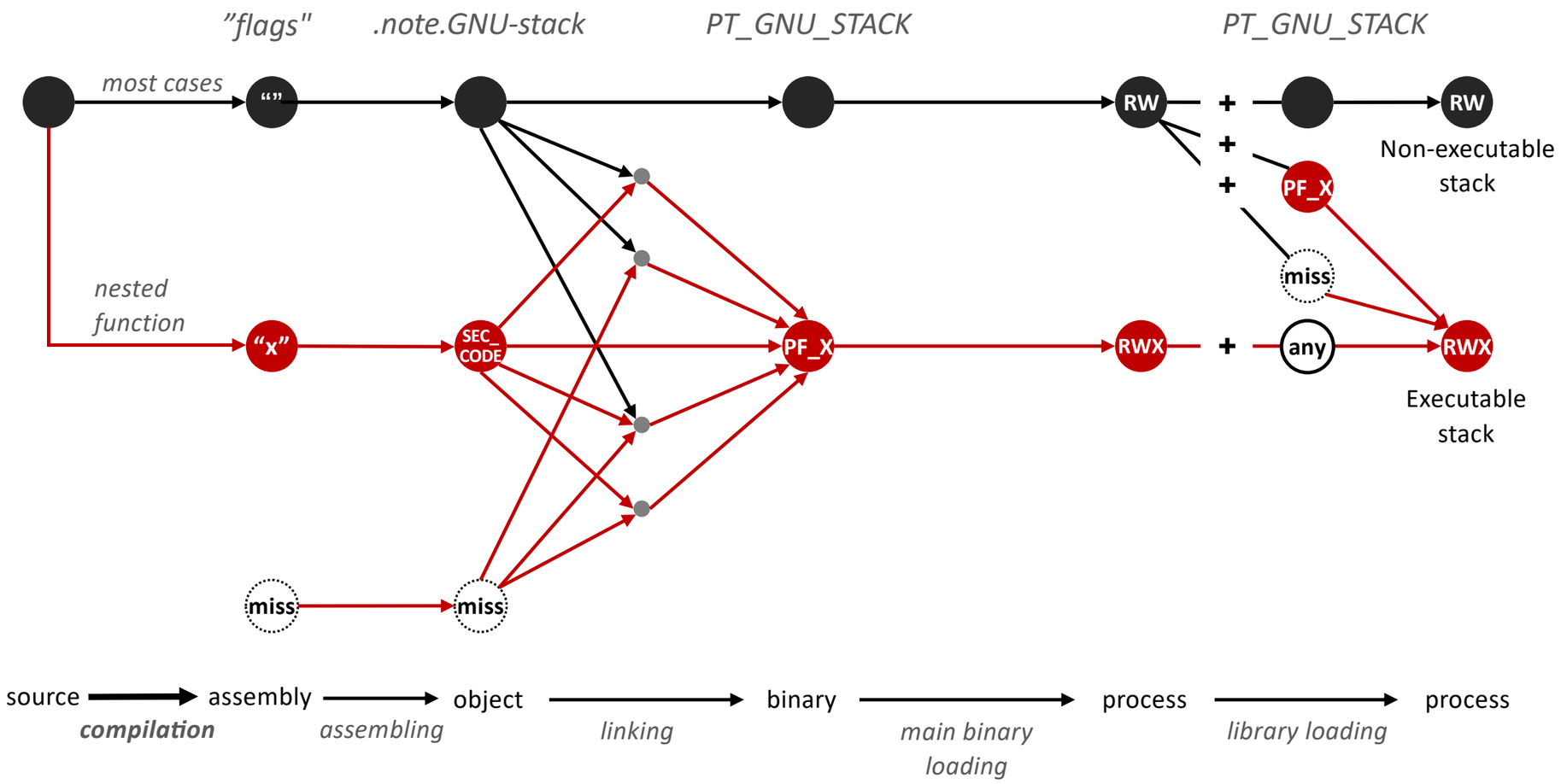


Stack Permission (Compiling)



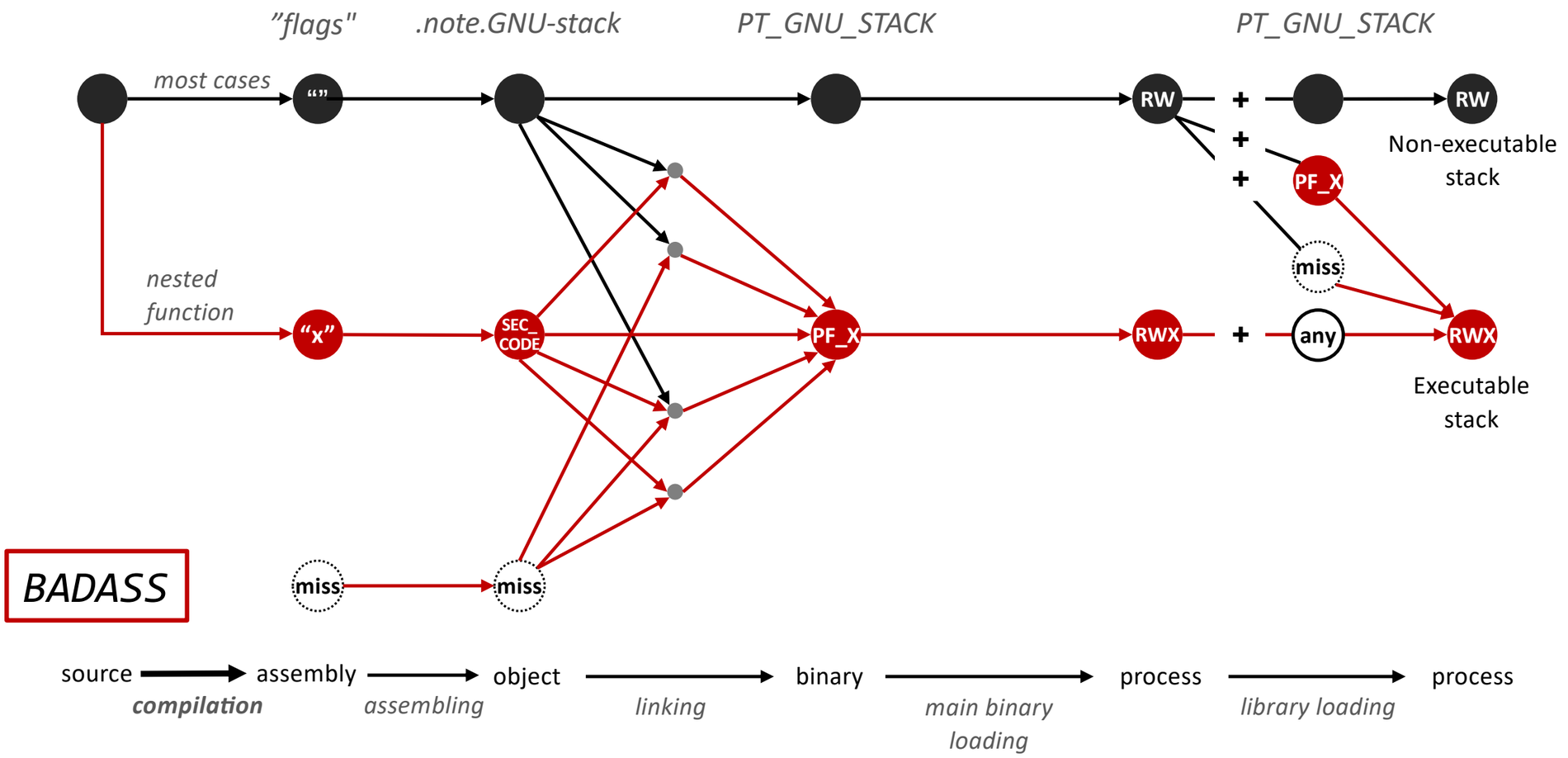
Stack Permission (Compiling)

- Compiler always produces `.section .note.GNU-stack,"flags”,@progbits`



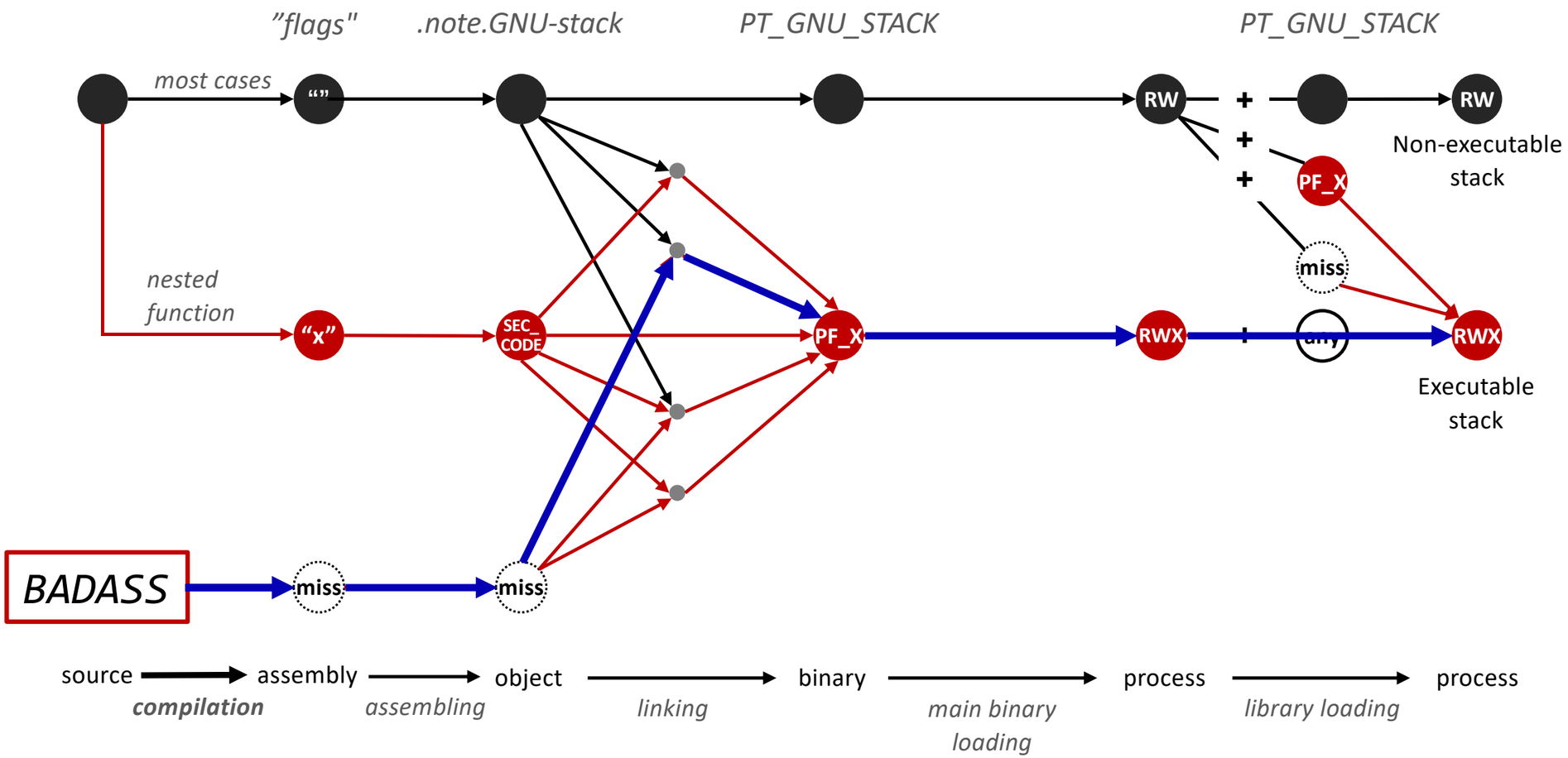
Stack Permission (Compiling)

- Compiler always produces `.section .note.GNU-stack,"flags”,@progbits`



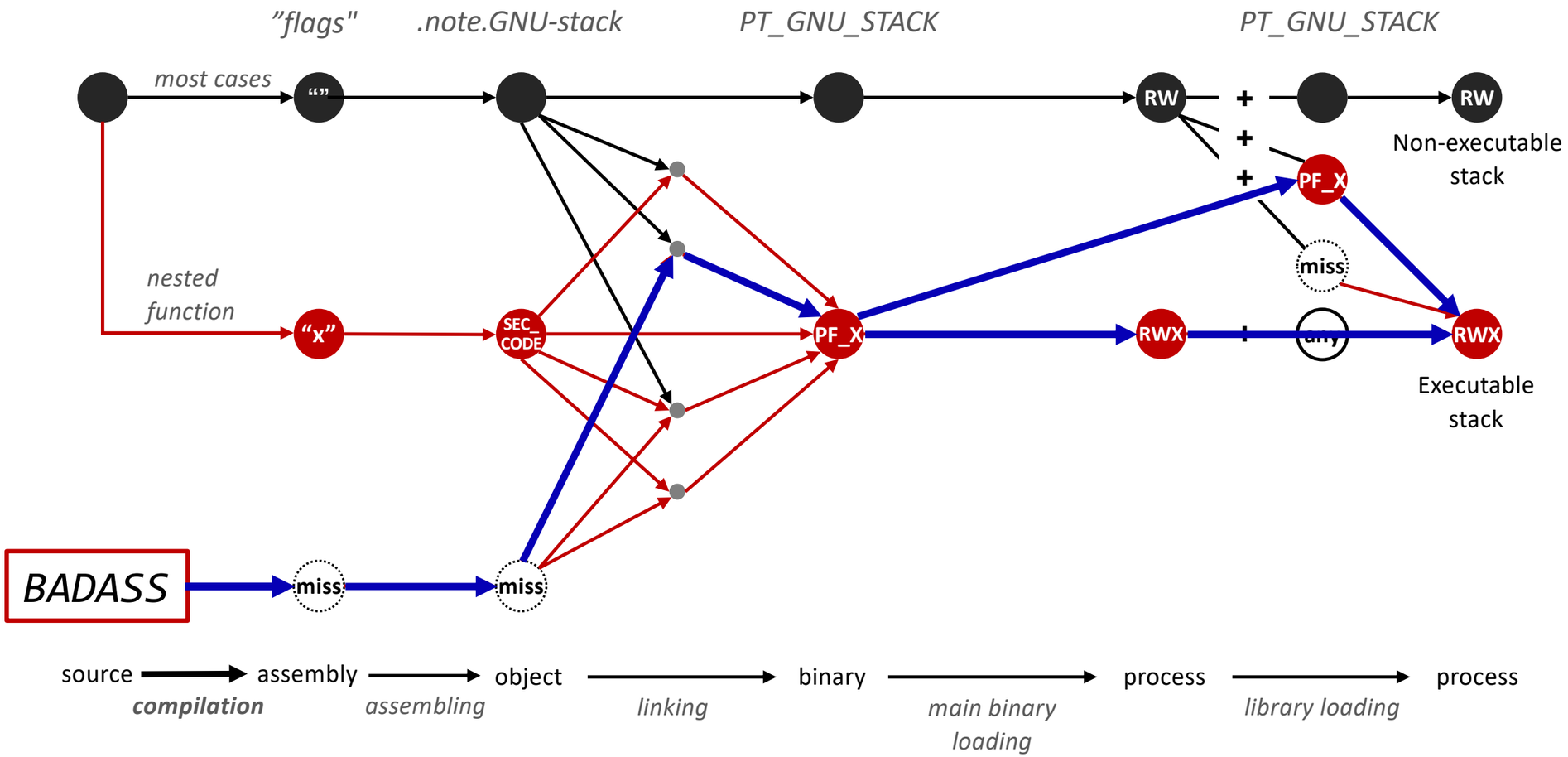
Stack Permission (Compiling)

- Compiler always produces `.section .note.GNU-stack,"flags”,@progbits`



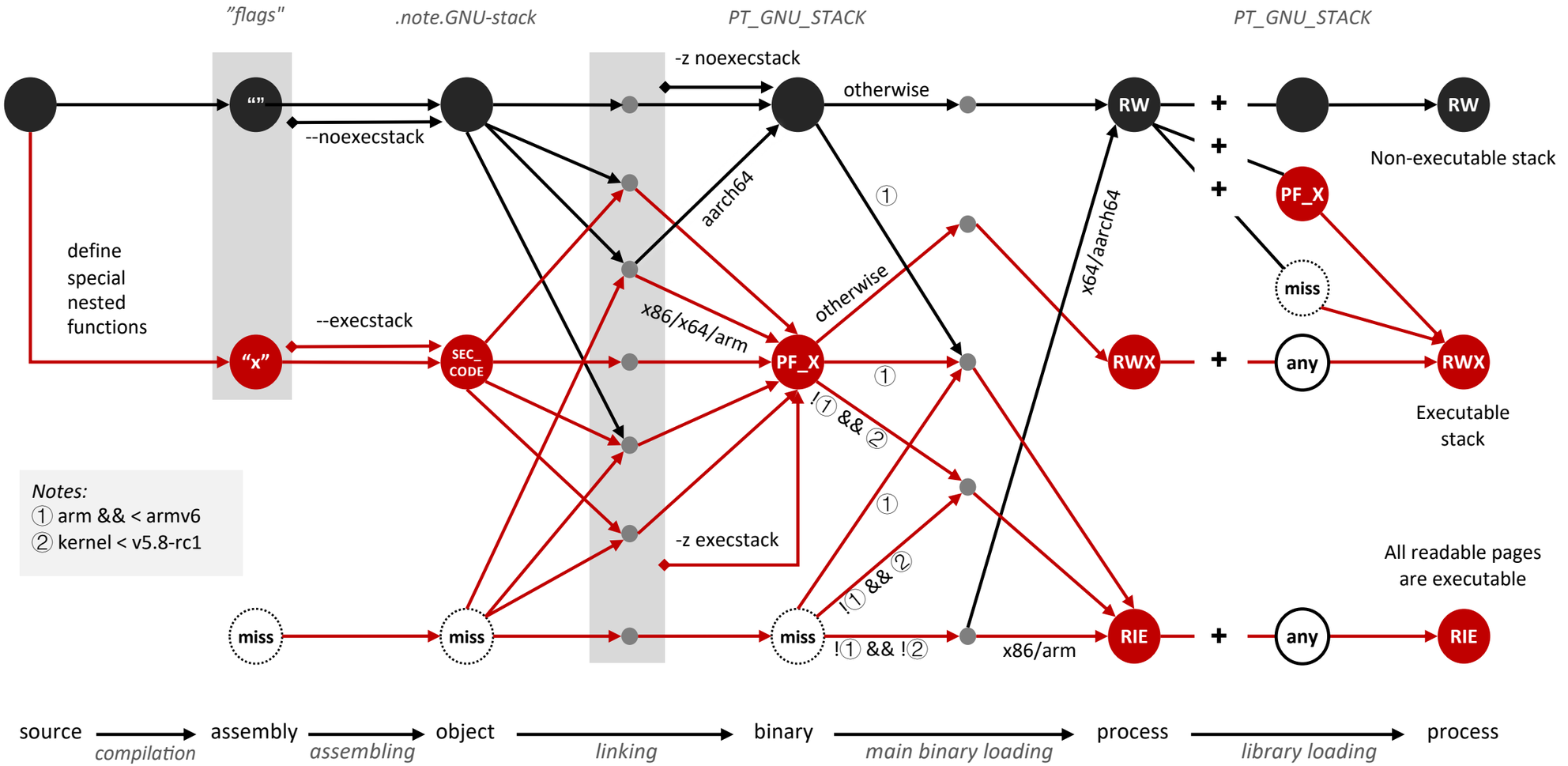
Stack Permission (Compiling)

- Compiler always produces `.section .note.GNU-stack,"flags”,@progbits`



Stack Permission (All Together)

- More complex details in [our paper]



Takeaway 2

In-depth analysis!

Subtle design

Developer

W⊕X

Takeaway 2

In-depth analysis!

Subtle design

Developer

W⊕X

- W⊕X enforcement on Linux requires concerted collaborations
  - Compiler, assembler, linker, loader, and kernel

Takeaway 2

In-depth analysis!

Subtle design

Developer

W⊕X

- W⊕X enforcement on Linux requires concerted collaborations
  - Compiler, assembler, linker, loader, and kernel
- Assembly files, not generated by compiler, break the regular enforcement routine

Takeaway 2

In-depth analysis!

Subtle design

Developer

W⊕X

- W⊕X enforcement on Linux requires concerted collaborations
  - Compiler, assembler, linker, loader, and kernel
- Assembly files, not generated by compiler, break the regular enforcement routine
- Subtlety of `.note.GNU-stack` section is the main root cause of executable stacks

Anything Else?

Anything Else?

- Missing `.section .note.gnu.property,”a”`

Anything Else?

- Missing `.section .note.gnu.property,”a”`
  - GNU_PROPERTY_X86_FEATURE_1_IBT
    - Disable Intel CET IBT

Anything Else?

- Missing `.section .note.gnu.property,”a”`
  - GNU_PROPERTY_X86_FEATURE_1_IBT
    - Disable Intel CET IBT
  - GNU_PROPERTY_X86_FEATURE_1_SHSTK
    - Disable Intel CET SHSTK

Anything Else?

- Missing `.section .note.gnu.property,”a”`
  - GNU_PROPERTY_X86_FEATURE_1_IBT
    - Disable Intel CET IBT
  - GNU_PROPERTY_X86_FEATURE_1_SHSTK
    - Disable Intel CET SHSTK
  - GNU_PROPERTY_STACK_SIZE
    - Trigger stack overflow

Anything Else?

- Missing `.section .note.gnu.property,”a”`
  - GNU_PROPERTY_X86_FEATURE_1_IBT
    - Disable Intel CET IBT
  - GNU_PROPERTY_X86_FEATURE_1_SHSTK
    - Disable Intel CET SHSTK
  - GNU_PROPERTY_STACK_SIZE
    - Trigger stack overflow
  - GNU_PROPERTY_NO_COPY_ON_PROTECTED
    - Evil Copy [1]

[1] Ge, Xinyang, Mathias Payer, and Trent Jaeger. "An Evil Copy: How the Loader Betrays You." NDSS. 2017.

Mitigations

Mitigations

- Disable executable stack by default
  - Only enable via explicit compilation option
    - “--execstack” to assembler **as**
    - “-z execstack” to linker **ld**

Mitigations

- Disable executable stack by default
  - Only enable via explicit compilation option
    - “--execstack” to assembler **as**
    - “-z execstack” to linker **ld**

- Always include the directive when using assembly
  - `.section .note.GNU-stack,””,@progbits`

Mitigations

- Disable executable stack by default
  - Only enable via explicit compilation option
    - “--execstack” to assembler **as**
    - “-z execstack” to linker **ld**

- Always include the directive when using assembly
  - `.section .note.GNU-stack,””,@progbits`

- Kernel/loader asks for user confirmation

Stack Permission on Other OSes

Stack Permission on Other OSes

- **OpenBSD**
  - Pioneer on W⊕X

Stack Permission on Other OSes

- **OpenBSD**
  - Pioneer on W⊕X
  - Mandatory since 2016

Stack Permission on Other OSes

- **OpenBSD**
  - Pioneer on W⊕X
  - Mandatory since 2016
- **Windows**
  - Not executable by default

Stack Permission on Other OSes

- **OpenBSD**
  - Pioneer on W⊕X
  - Mandatory since 2016
- **Windows**
  - Not executable by default
  - Executable w/
    - .def __enable_execute_stack
    - call __enable_execute_stack

Stack Permission on Other OSes

- **OpenBSD**
  - Pioneer on W⊕X
  - Mandatory since 2016
- **Windows**
  - Not executable by default
  - Executable w/
    - .def __enable_execute_stack
    - call __enable_execute_stack
- **macOS**
  - Intel CPU
    - Linker option: -allow_stack_execute
    - Program header: ALLOW_STACK_EXECUTION

Stack Permission on Other OSes

- **OpenBSD**
  - Pioneer on W⊕X
  - Mandatory since 2016
- **Windows**
  - Not executable by default
  - Executable w/
    - .def __enable_execute_stack
    - call __enable_execute_stack
- **macOS**
  - Intel CPU
    - Linker option: -allow_stack_execute
    - Program header: ALLOW_STACK_EXECUTION
  - Apple Silicon
    - Always not executable

Conclusion

- *BADASS* in inlined reference monitor
  - 11/21 have *BADASS* issue
  - Even security experts can not avoid

Conclusion

- *BADASS* in inlined reference monitor
  - 11/21 have *BADASS* issue
  - Even security experts can not avoid
- In-depth analysis of W⊕X enforcement
  - Concerted collaborations
  - Too subtle to notice

Conclusion

- *BADASS* in inlined reference monitor
  - 11/21 have *BADASS* issue
  - Even security experts can not avoid
- In-depth analysis of W⊕X enforcement
  - Concerted collaborations
  - Too subtle to notice
- Open source
  - https://github.com/psu-security-universe/badass

Thank You

Question?

hengkai@psu.edu