Surgically returning to randomized lib(c)

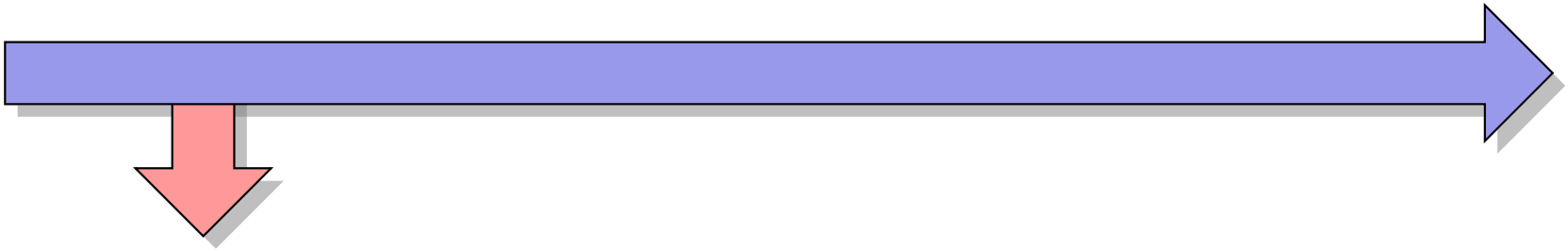
Giampaolo Fresi Roglia[1], Lorenzo Martignoni[2],
Roberto Paleari[1], Danilo Bruschi[1]

[1]Università degli Studi di Milano,
[2]Università degli Studi di Udine

December 9, 2009

Annual Computer Security Applications Conference (ACSAC)

Timeline of code injection attacks & defenses

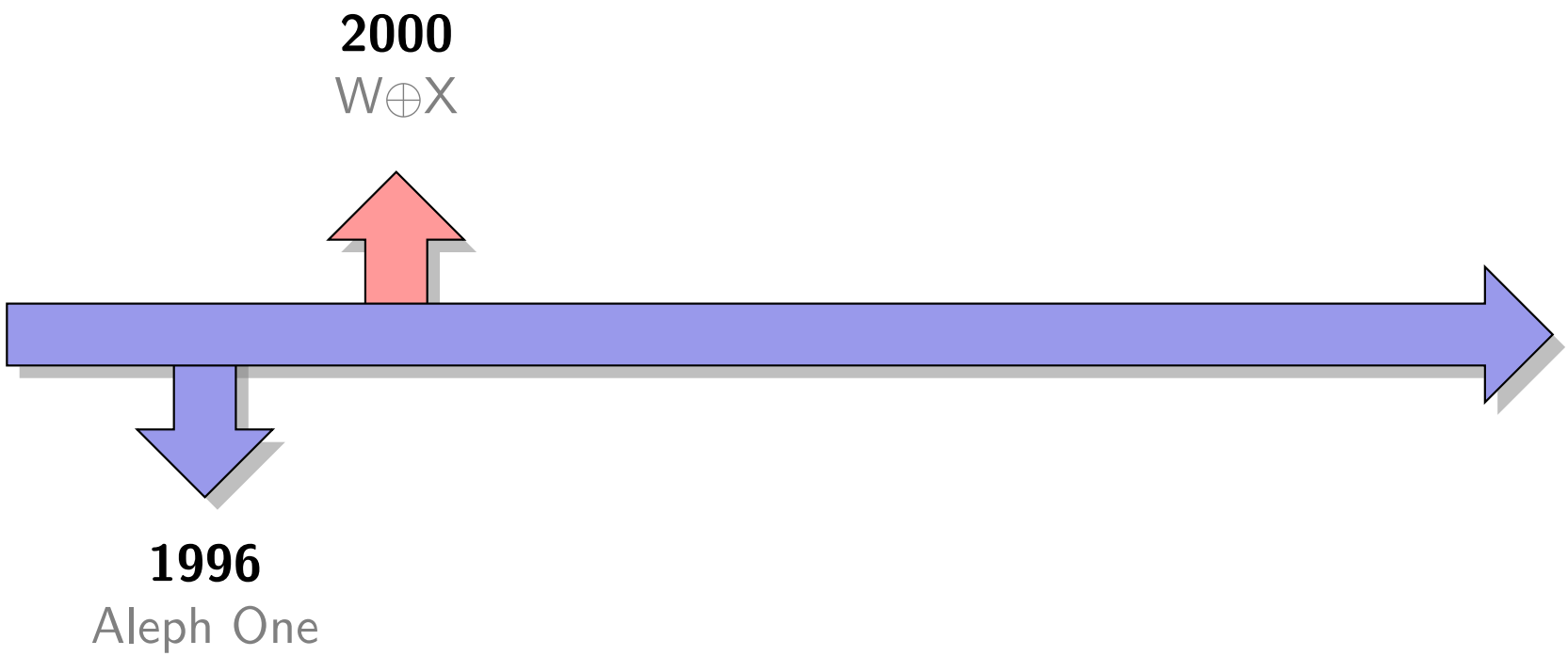


1996

Aleph One

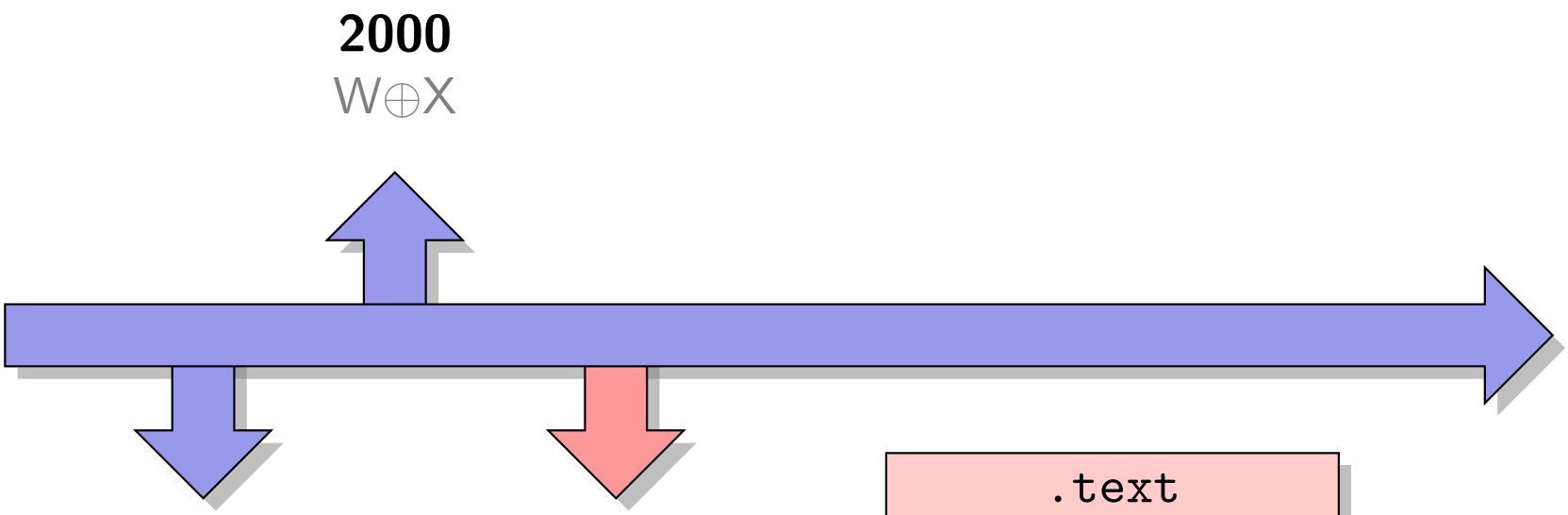
Smashing the stack for fun and profit, Aleph One, Phrack #49

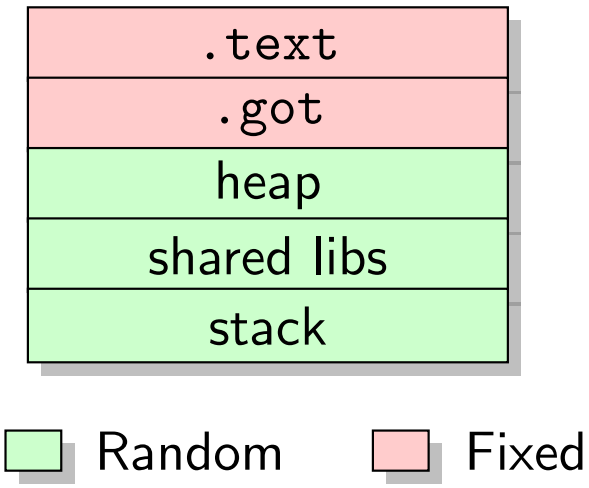
Timeline of code injection attacks & defenses



Prevent execution of writable memory locations
(code injected into the stack cannot be executed)

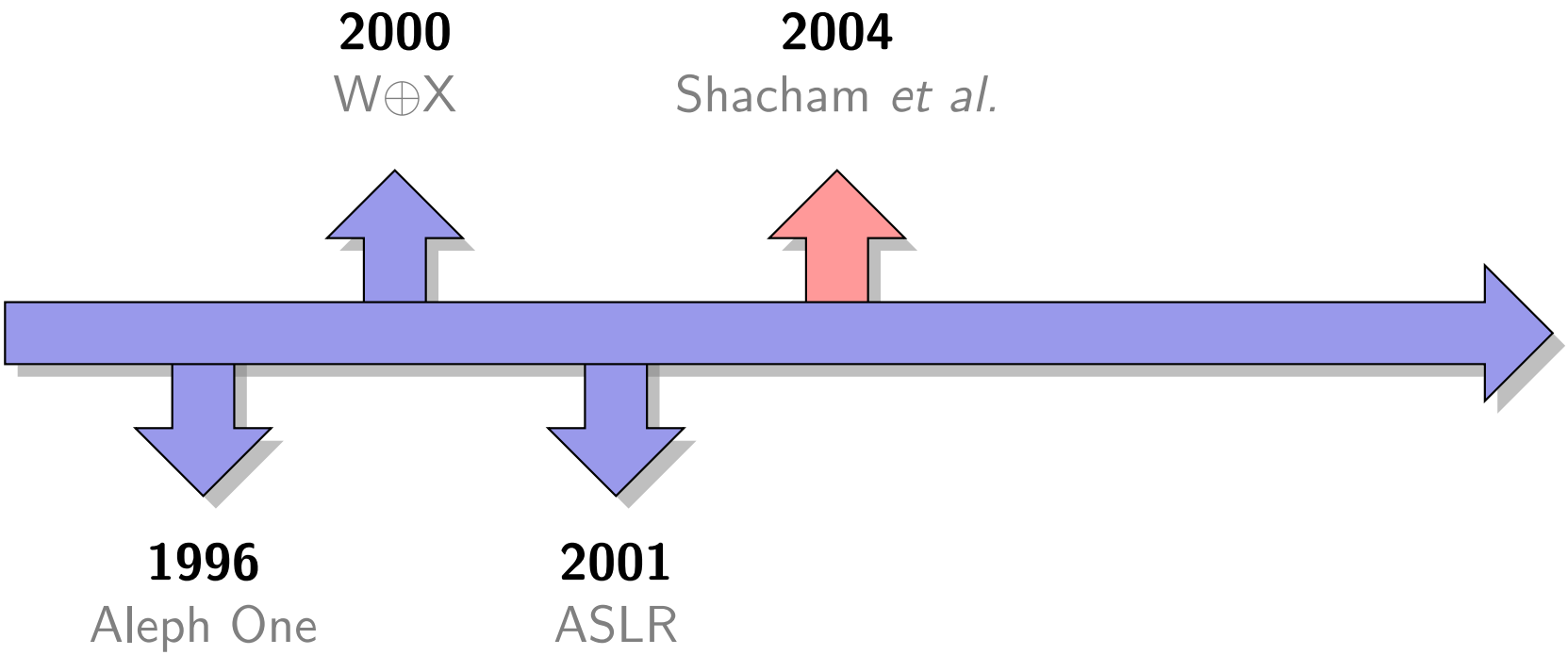
Timeline of code injection attacks & defenses





Stack, heap and shared libraries
base addresses are chosen randomly

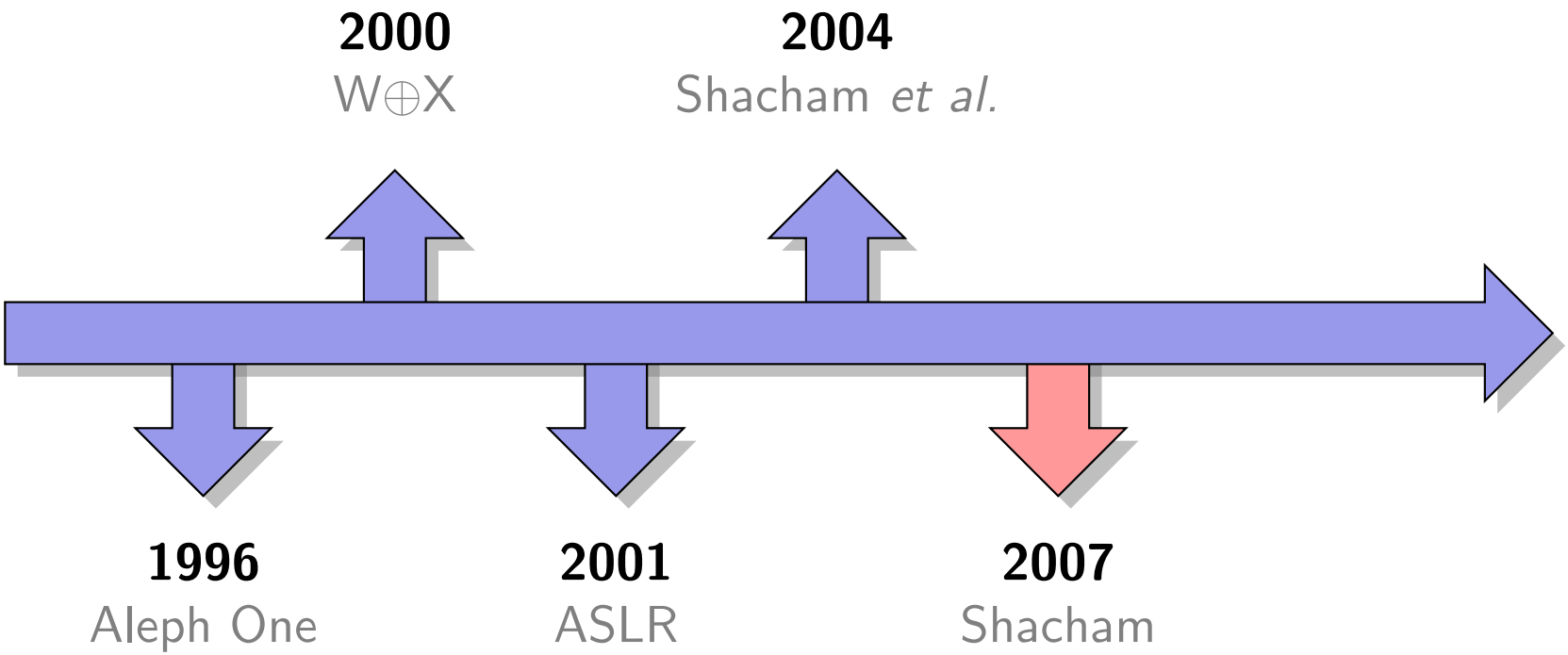
Timeline of code injection attacks & defenses



ASLR is sensitive to bruteforce: $\leq 2^{16}$ attempts on x86

On the effectiveness of address-space randomization, Shacham *et al.*, CCS 2004

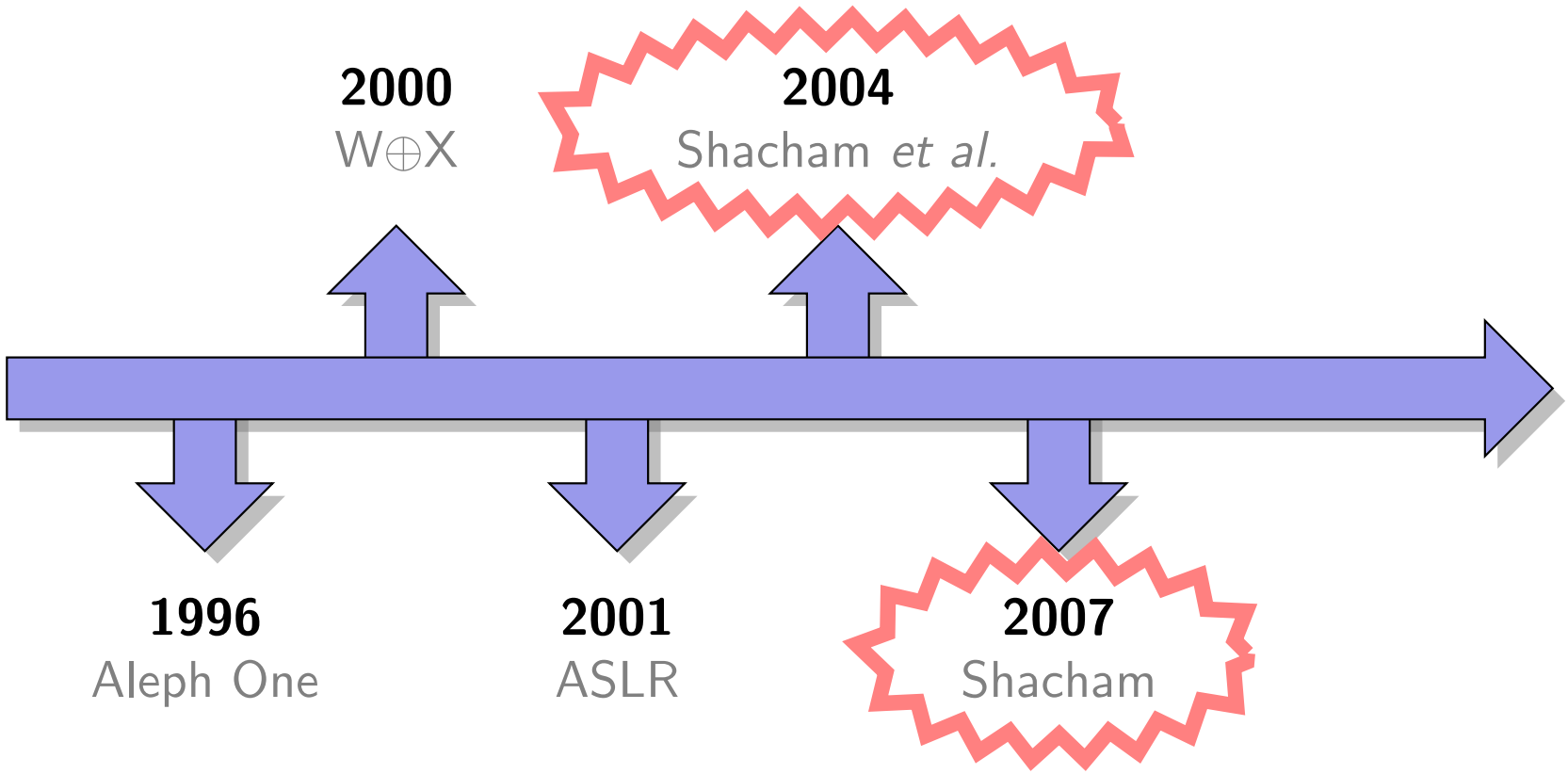
Timeline of code injection attacks & defenses



Perform Turing-complete computations
with return-oriented programming

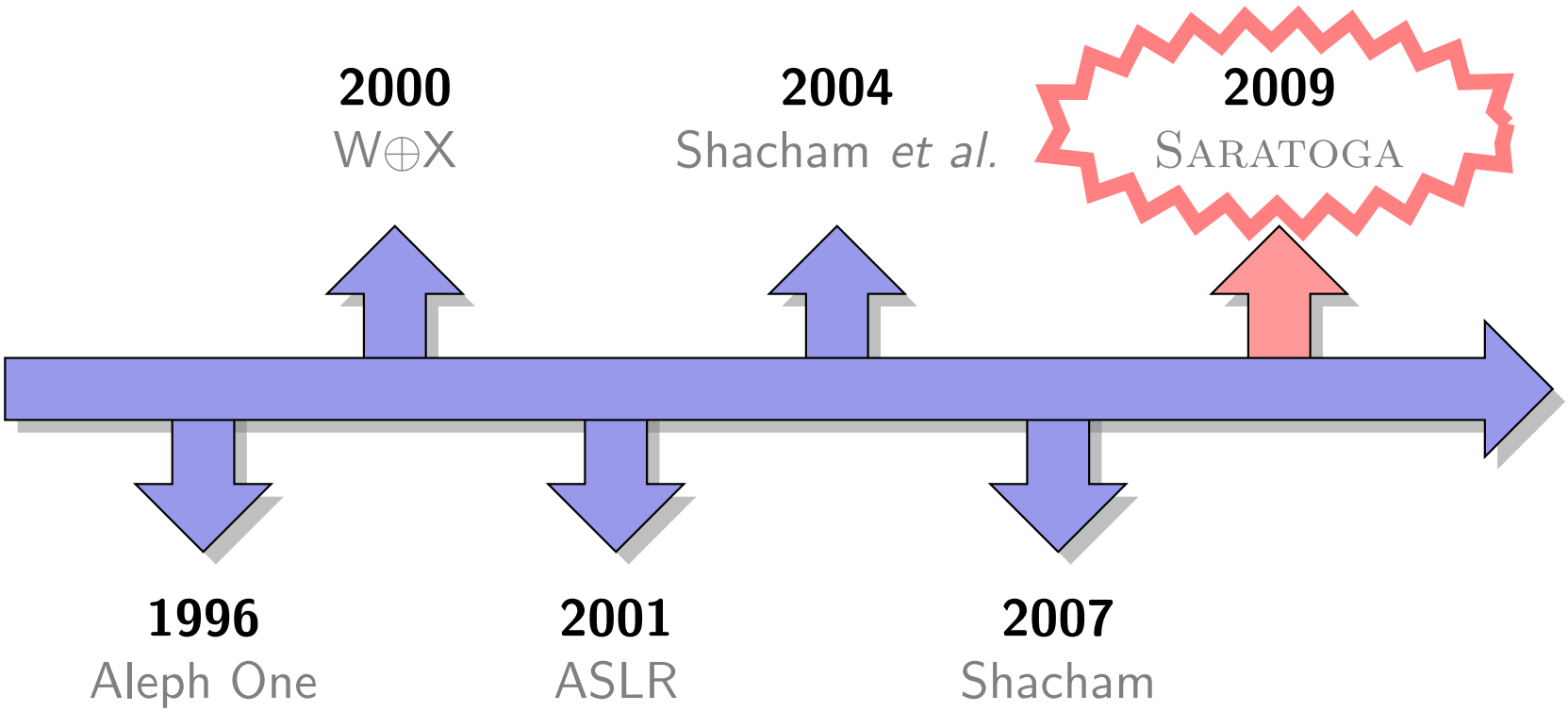
The geometry of innocent flesh on the bone, Shacham, CCS 2007

Timeline of code injection attacks & defenses



Attackers could bypass ASLR **and** W⊕X using brute force

Timeline of code injection attacks & defenses



New attack to bypass ASLR **and** W⊕X **surgically**
(i.e., in a single attempt)

Our contribution

Context

- No code injection allowed (W$\oplus$X)
- Classical return-into-libc is not possible (ASLR)

Contributions

- A new surgical exploitation technique for stack-based buffer overflows
- SARATOGA: a tool for automatic exploitation
- A new protection scheme that does not require recompilation and introduces a minimal run-time overhead (comparable to PIE)

The technique at a glance

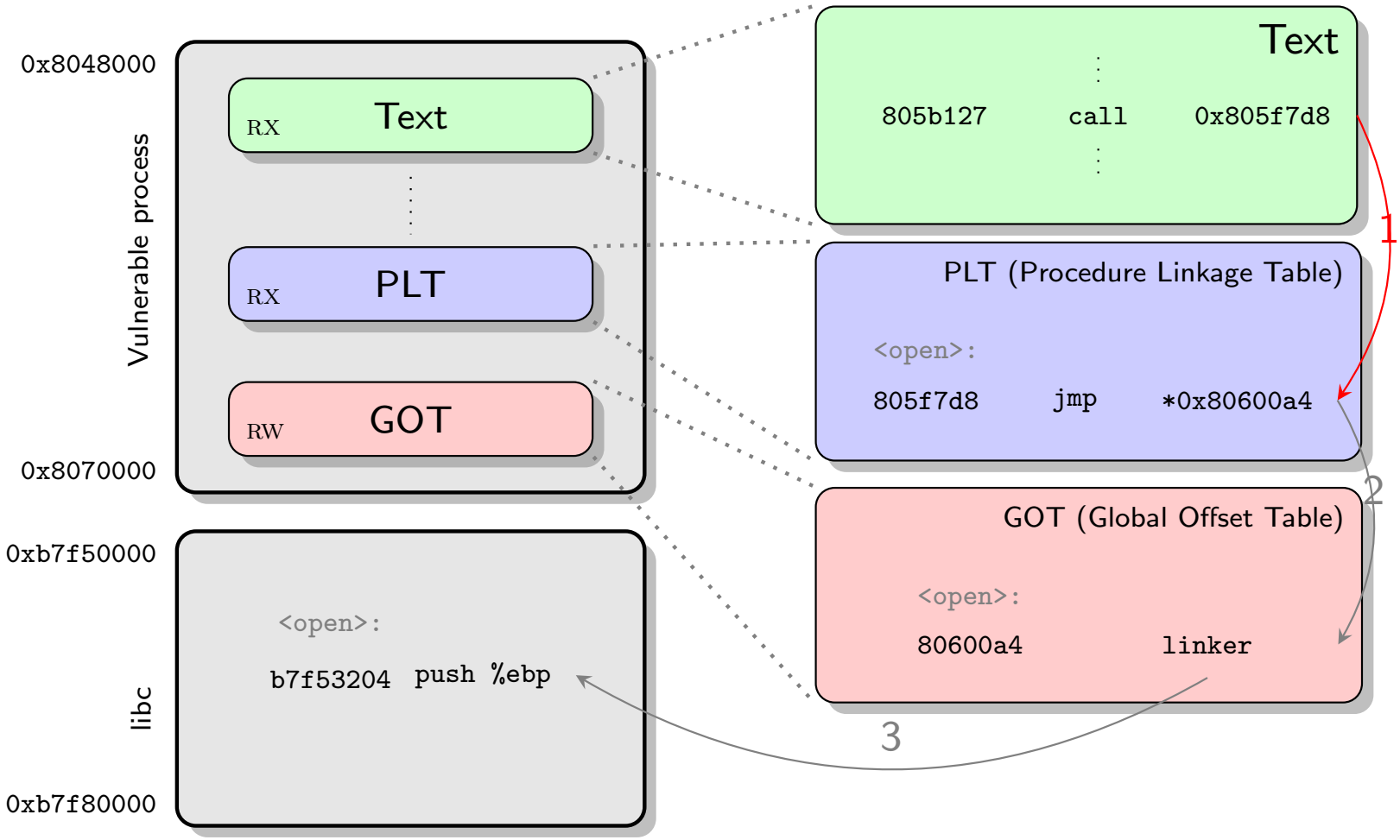
Information leakage

We exploit information about the base address of the lib(c), directly available in the memory of the process

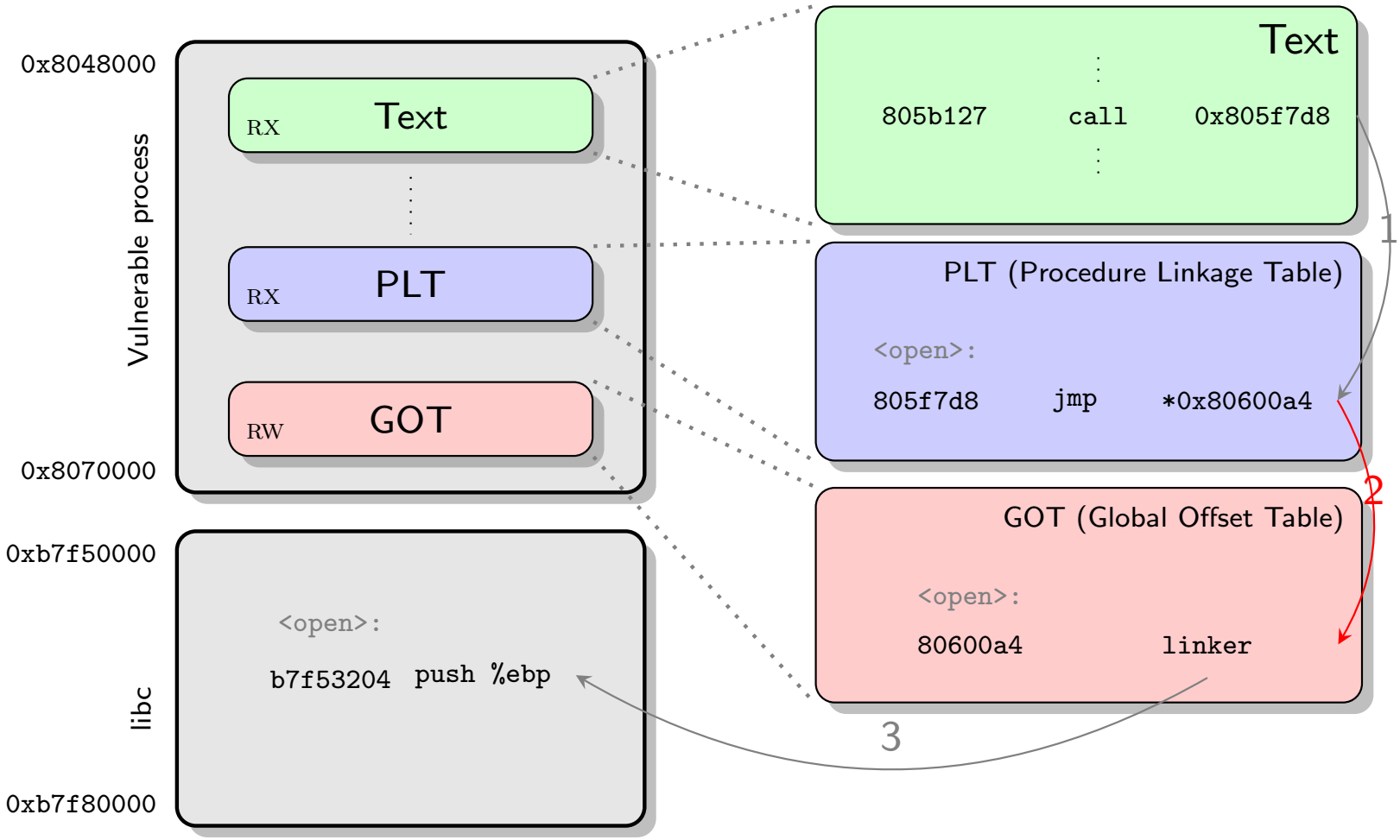
Code recycling

We combine few code fragments available at fixed addresses and use these fragments to discover the address of another libc function.

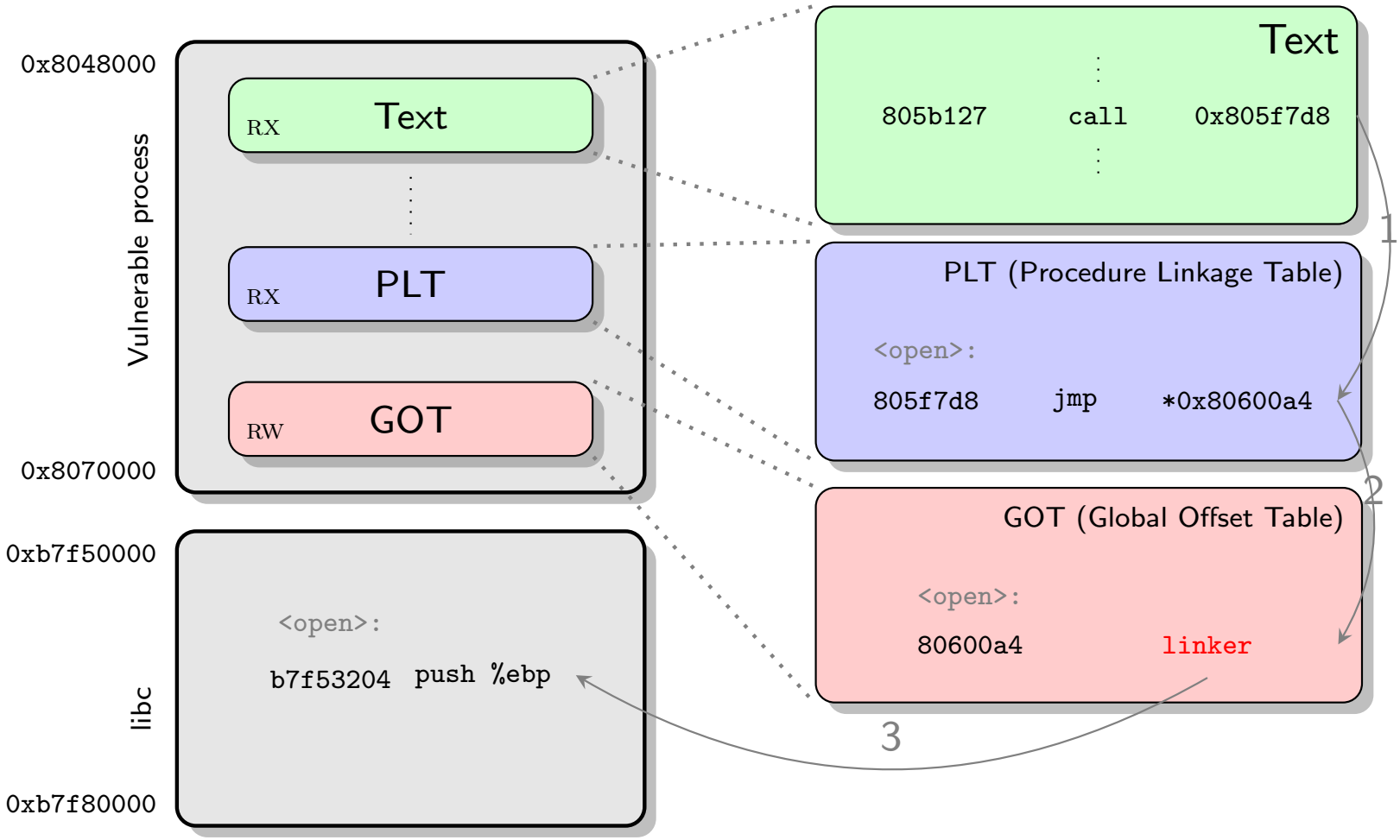
How function invocation works



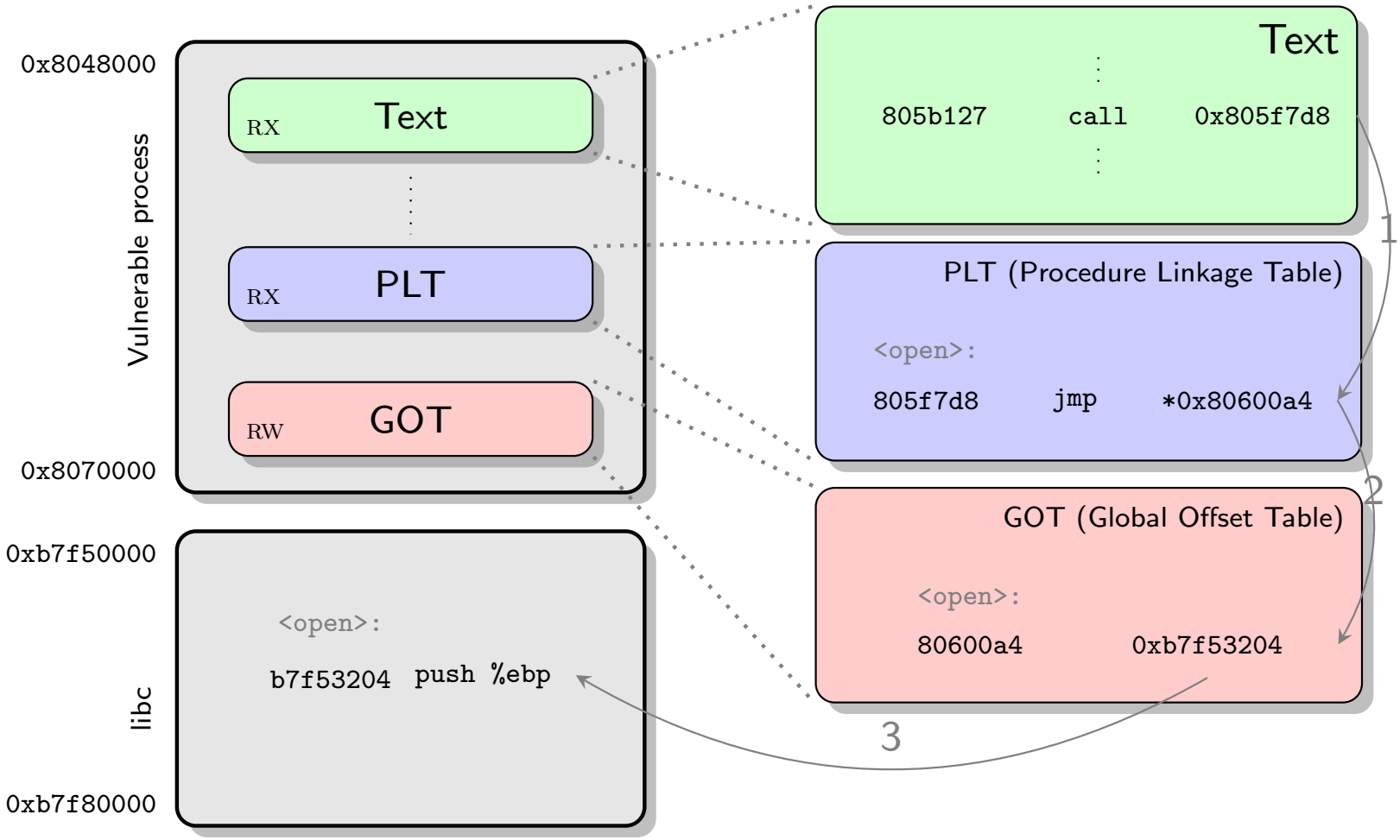
How function invocation works



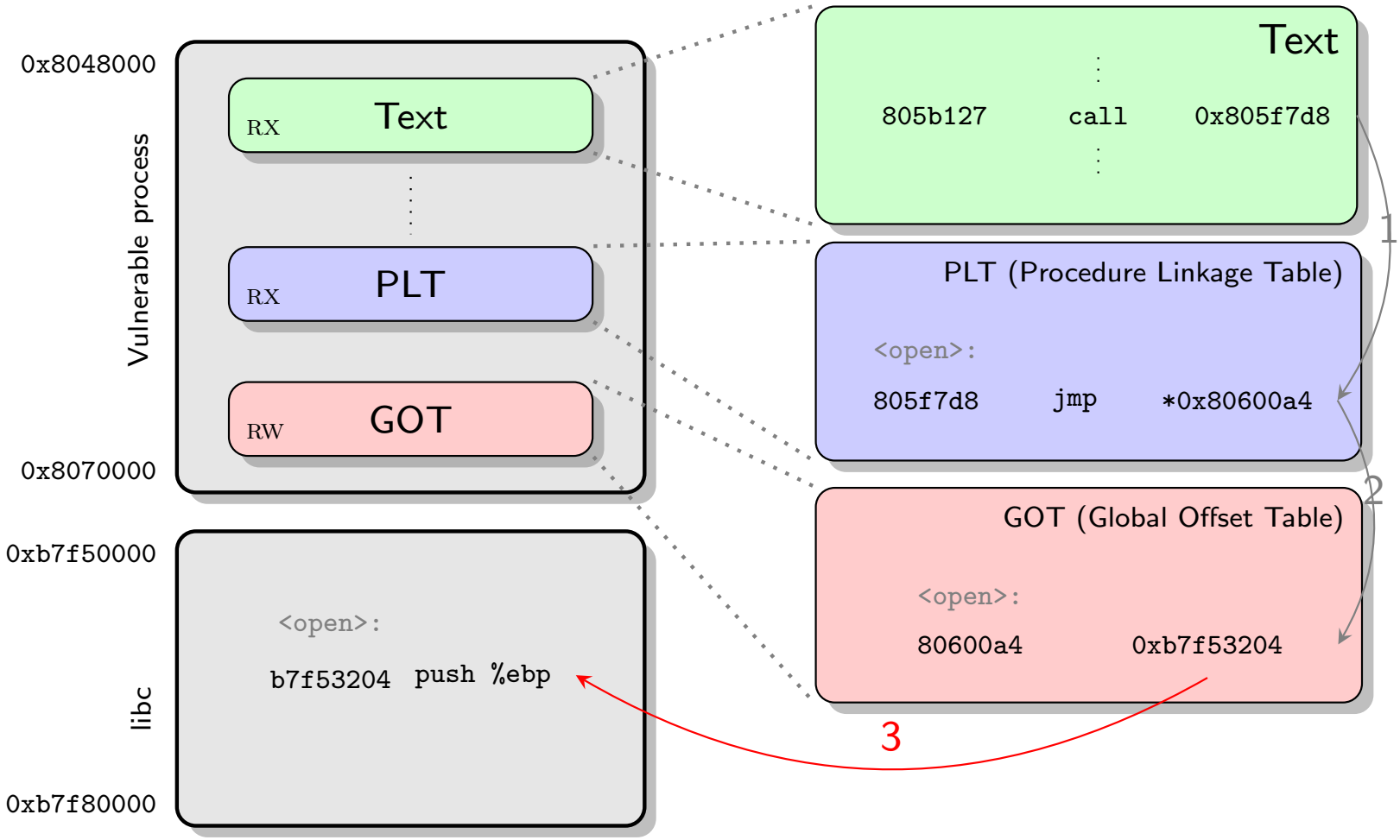
How function invocation works



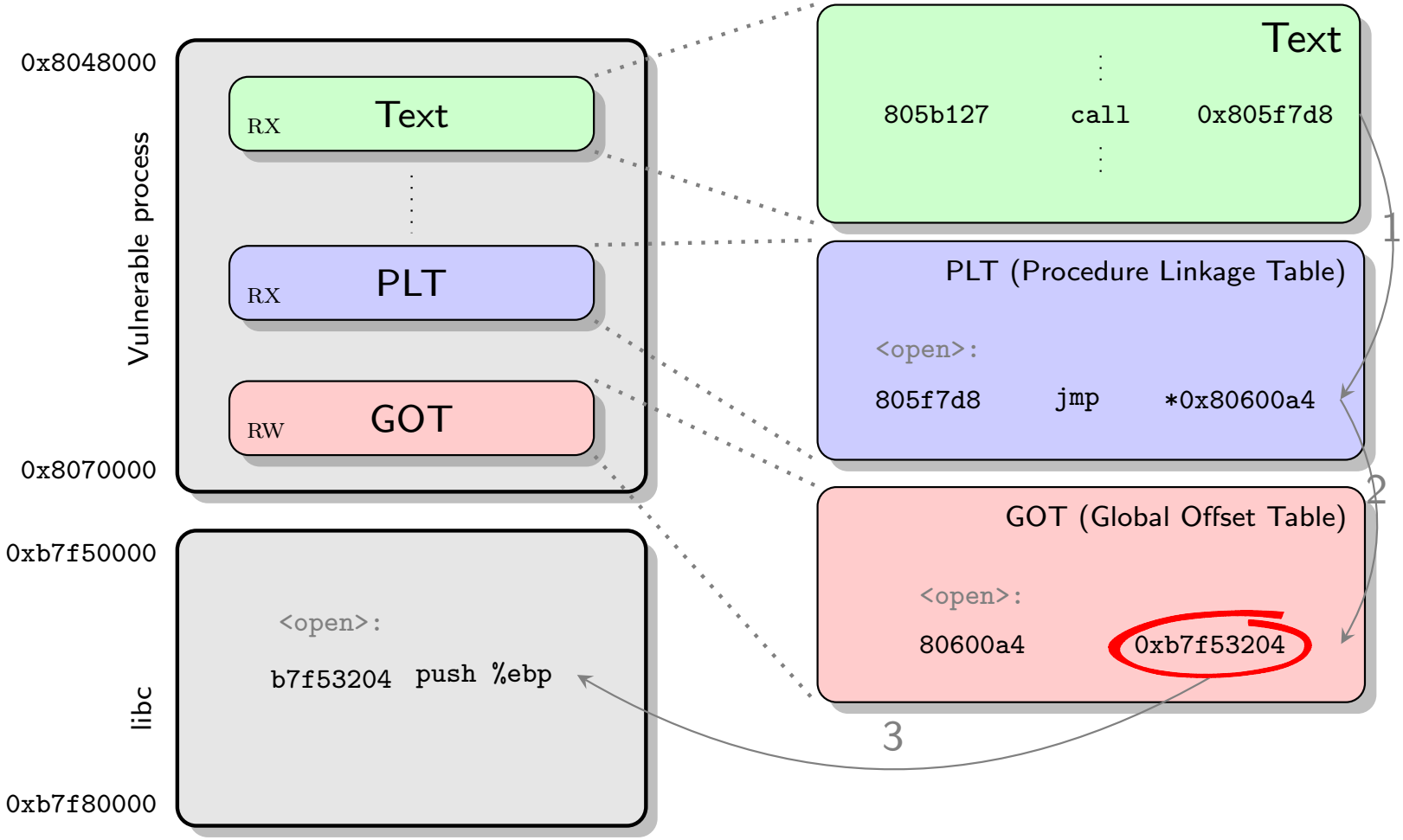
How function invocation works



How function invocation works



The leakage


0x8048000
Vulnerable process
RX
Text
RX
PLT
RW
GOT
0x8070000
0xb7f50000
libc
<open>:
b7f53204 push %ebp
0xb7f80000
Text
805b127 call 0x805f7d8
PLT (Procedure Linkage Table)
<open>:
805f7d8 jmp *0x80600a4
GOT (Global Offset Table)
<open>:
80600a4 0xb7f53204
1
2
3

How the leakage can be abused

Offsets between functions in libc are constants

address of(system) - address of(open) = constant

How the leakage can be abused

Offsets between functions in libc are constants

address of(system) = address of(open) + constant

How the leakage can be abused

We must let the process do the sum for us

How the leakage can be abused

We must let the process do the sum for us

How?

How the leakage can be abused

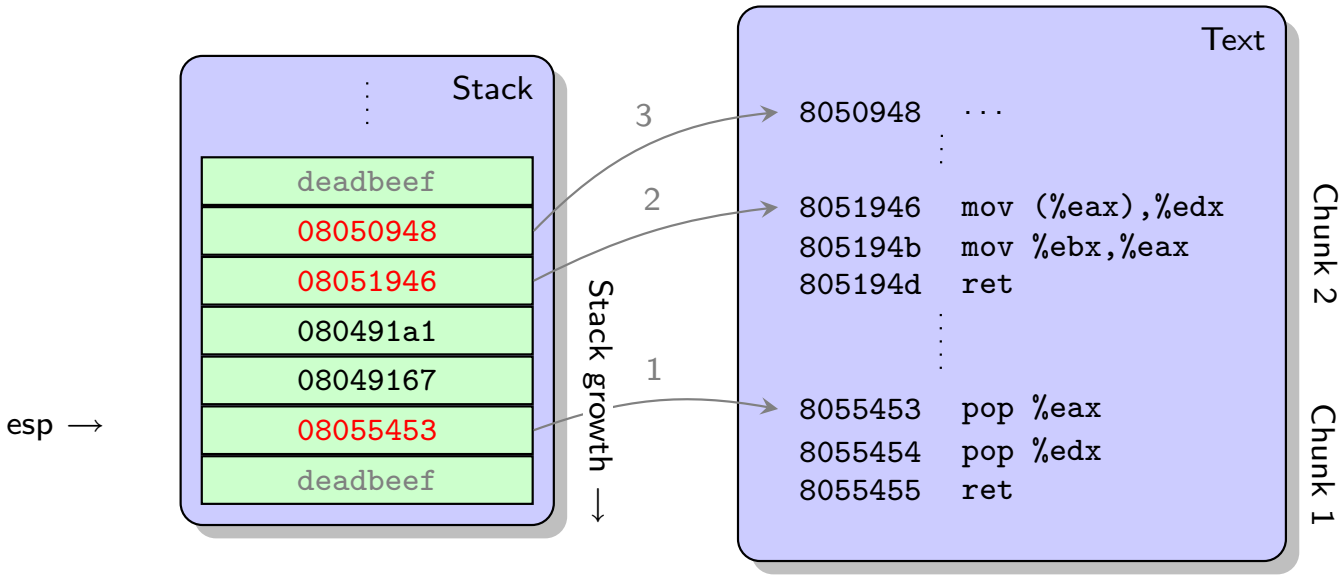
We must let the process do the sum for us

How?

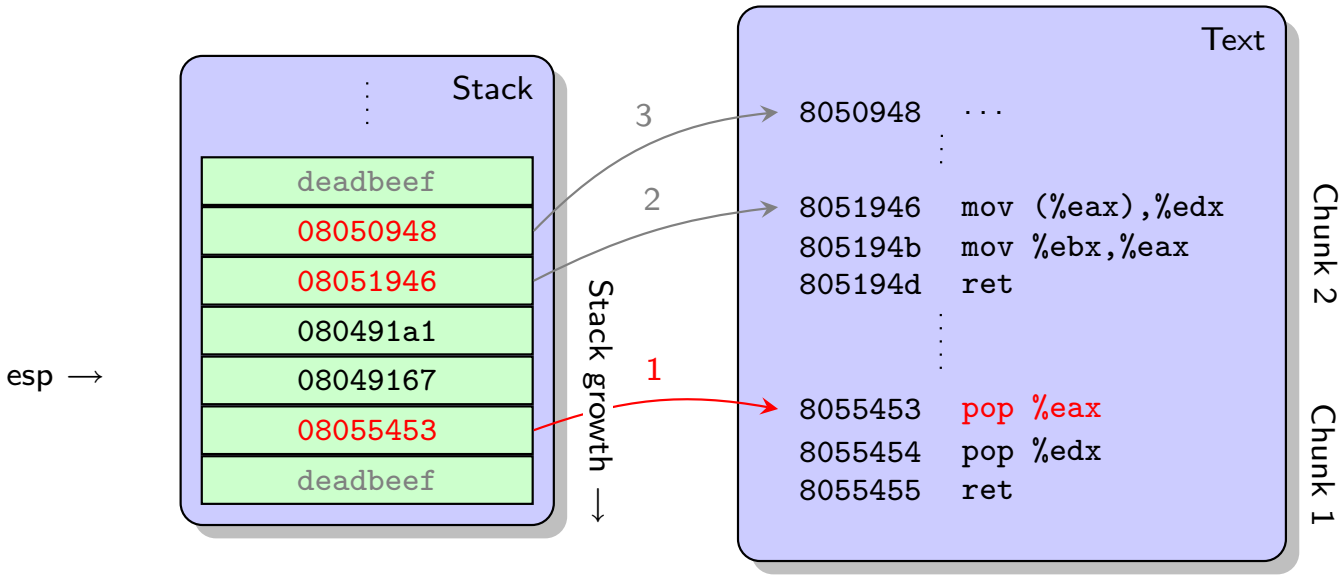
Return oriented programming

- Short code snippets ending with a `ret` instruction
- Each of them can accomplish simple tasks
- Can be glued together building a correct stack layout
- Glued together can accomplish more complex computations

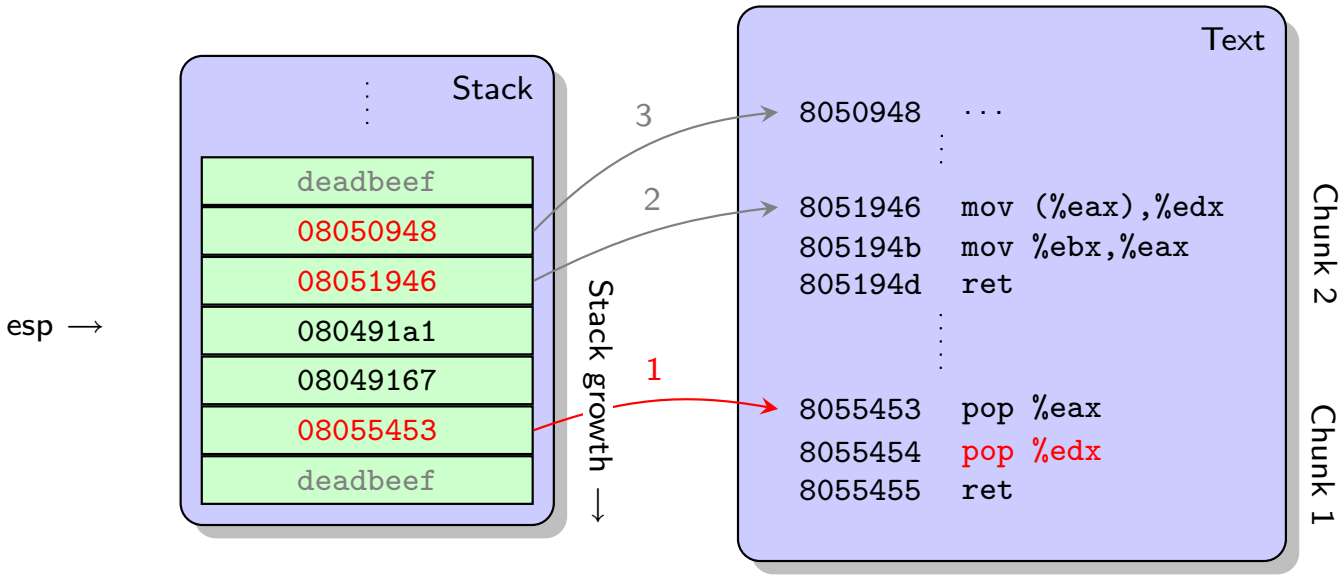
Return oriented programming



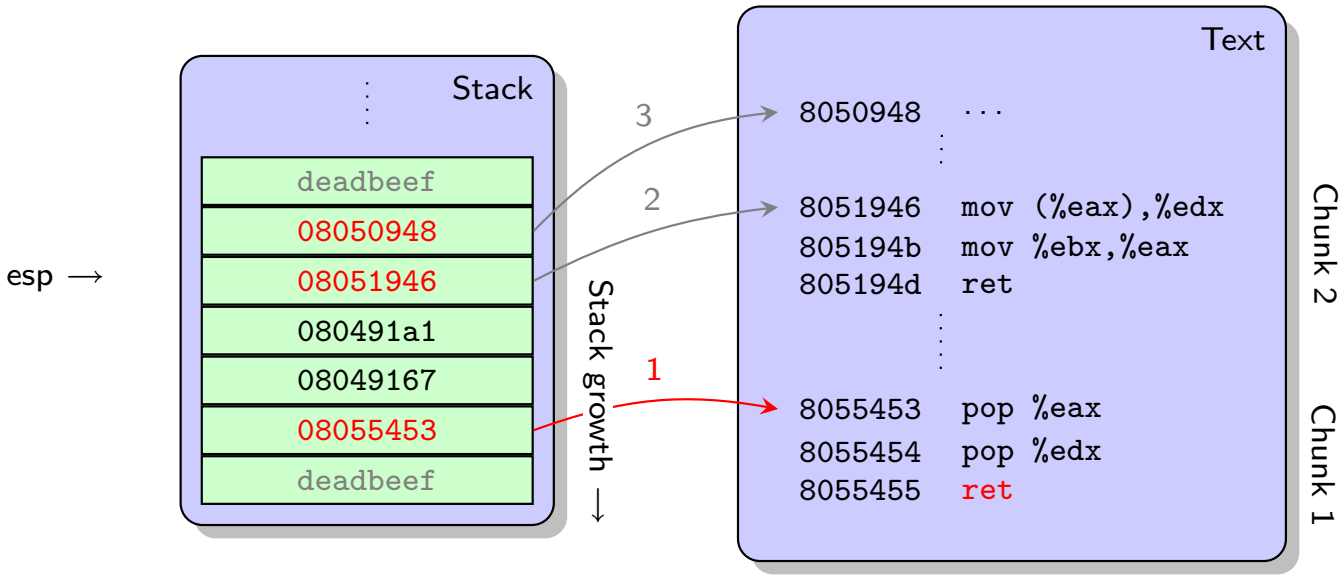
Return oriented programming



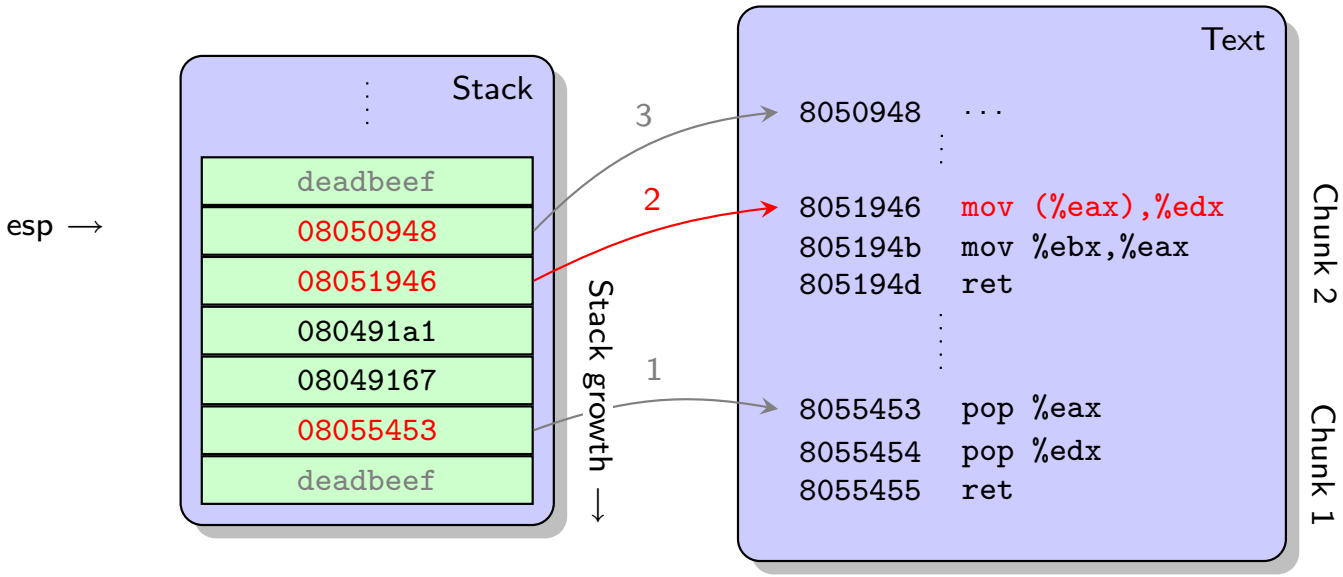
Return oriented programming



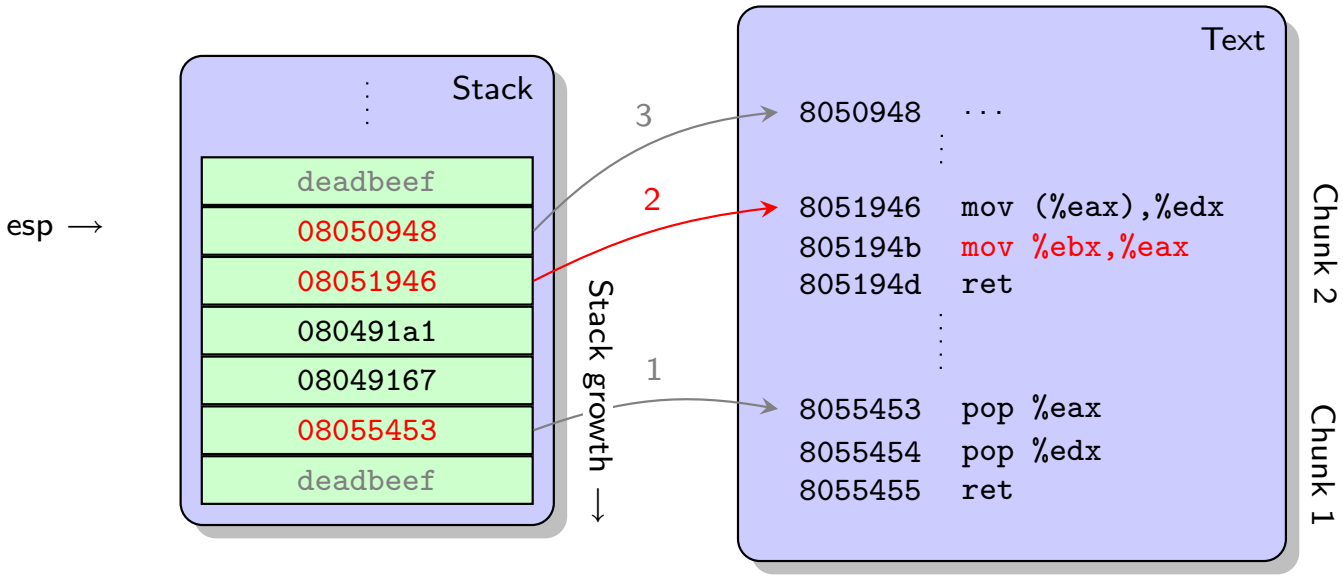
Return oriented programming



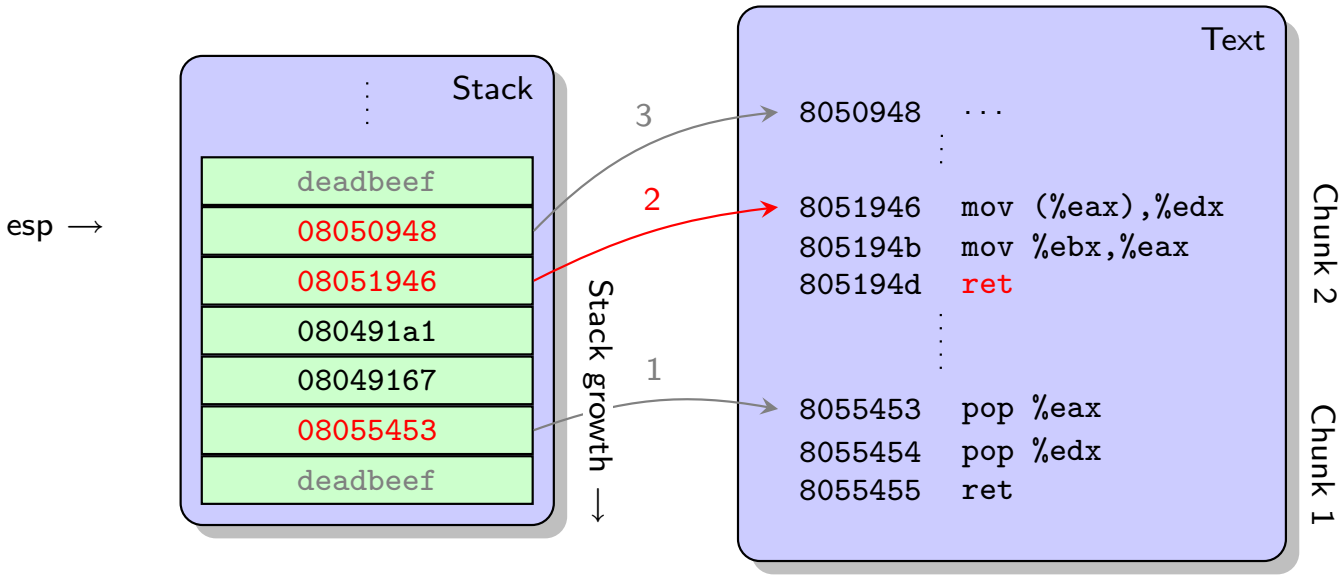
Return oriented programming



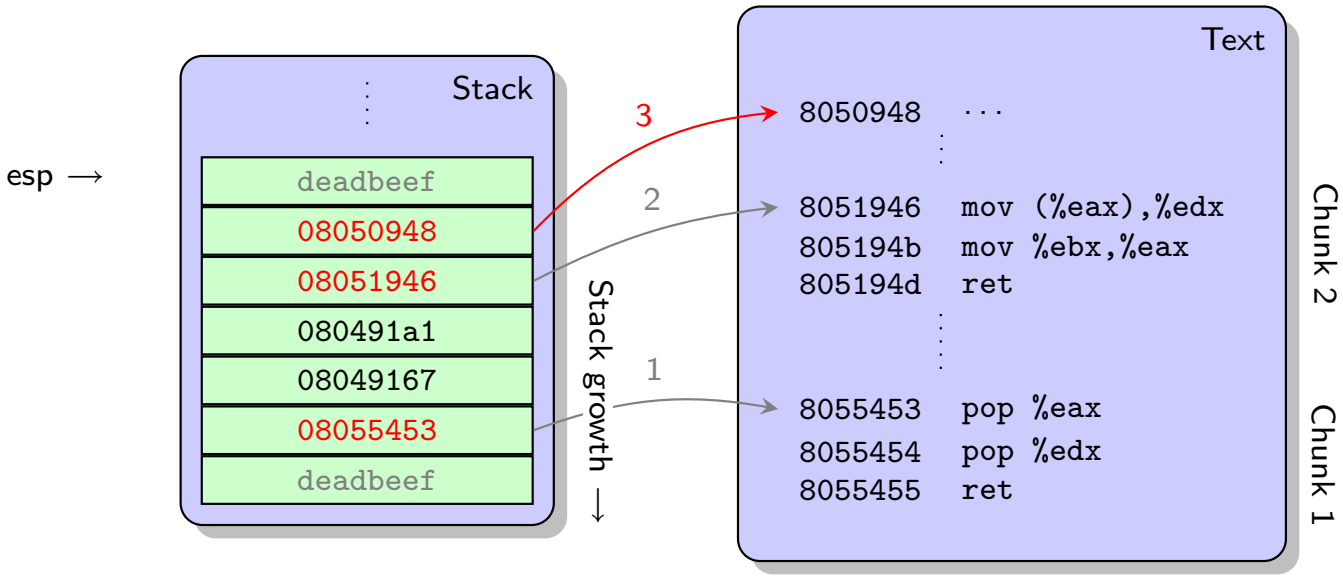
Return oriented programming



Return oriented programming



Return oriented programming



Return to randomized lib(c)

Return to randomized lib(c)

Two variants

- Got dereferencing
  - More powerful
  - Less likely to succeed
- Got overwriting
  - Less powerful
  - Most likely to succeed
  - Read-only GOT prevents it

Return to randomized lib(c)

Two variants

- Got dereferencing
  - More powerful
  - Less likely to succeed
- Got overwriting
  - Less powerful
  - Most likely to succeed
  - Read-only GOT prevents it

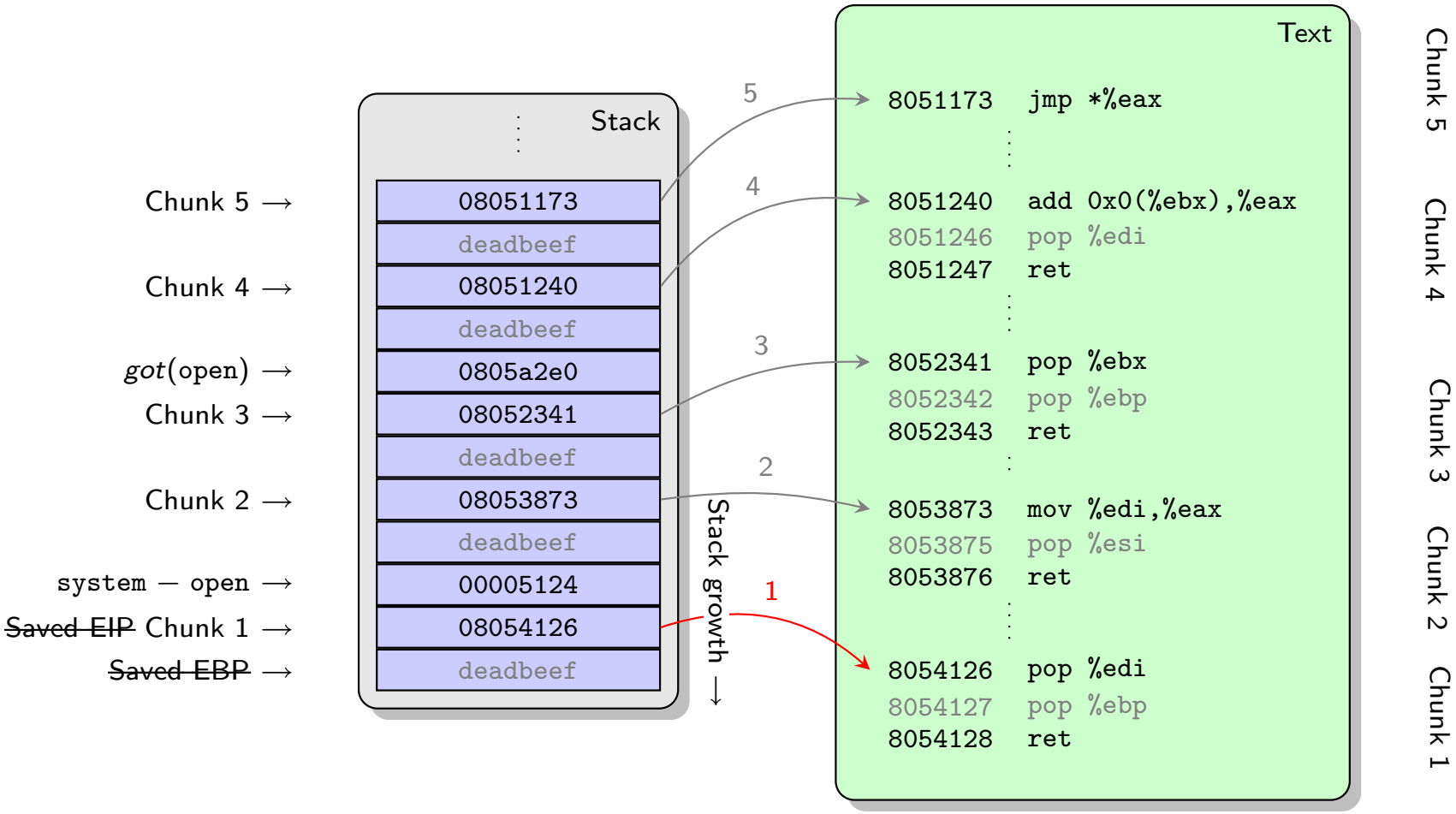
Got dereferencing

Got dereferencing

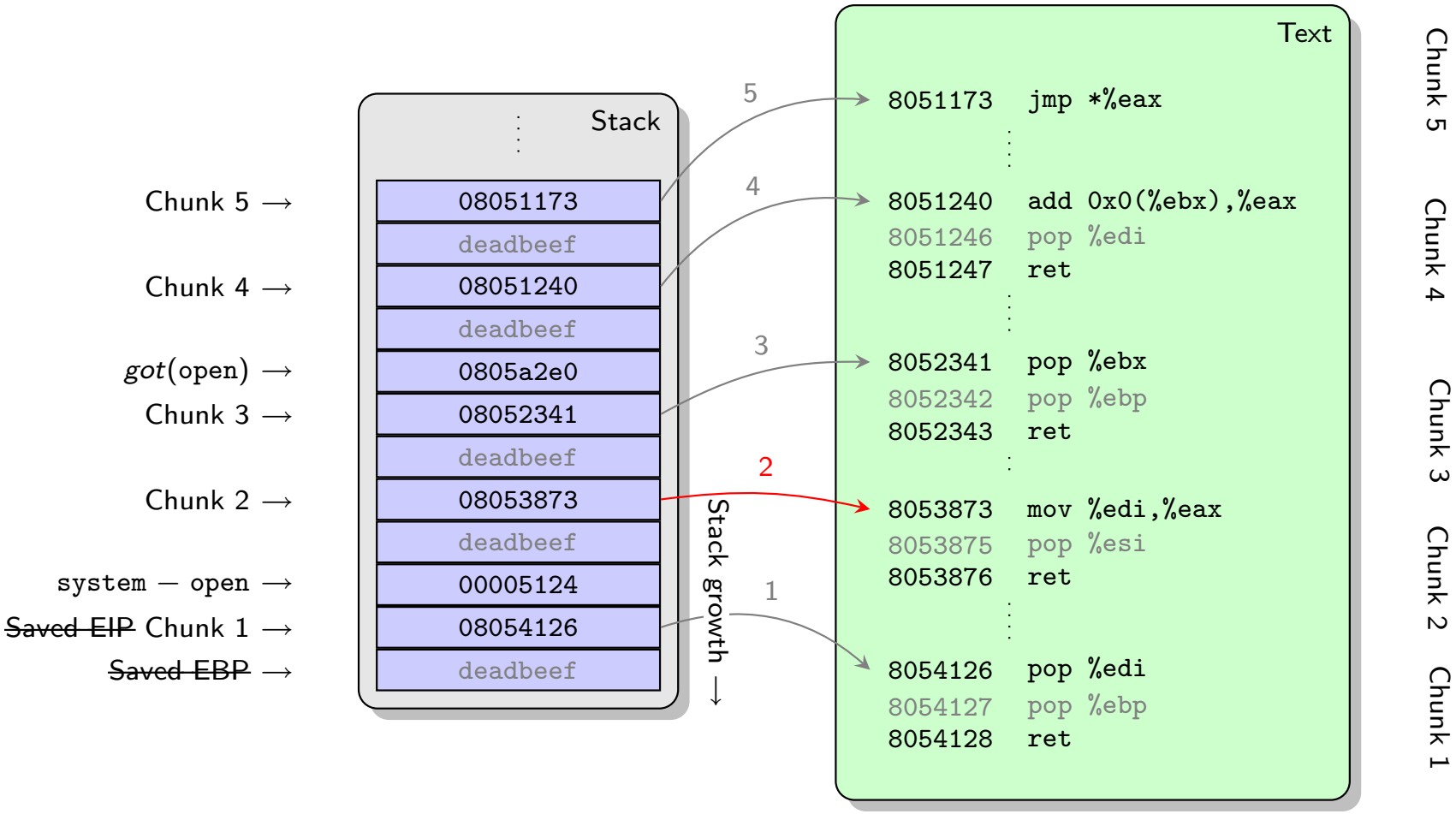
- Read the absolute address of a function (a GOT entry)
- Compute the absolute address of another function (e.g., system = open + *offset*)
- Store result to register
- Jump to the result

. . . all that by combining multiple code chunks

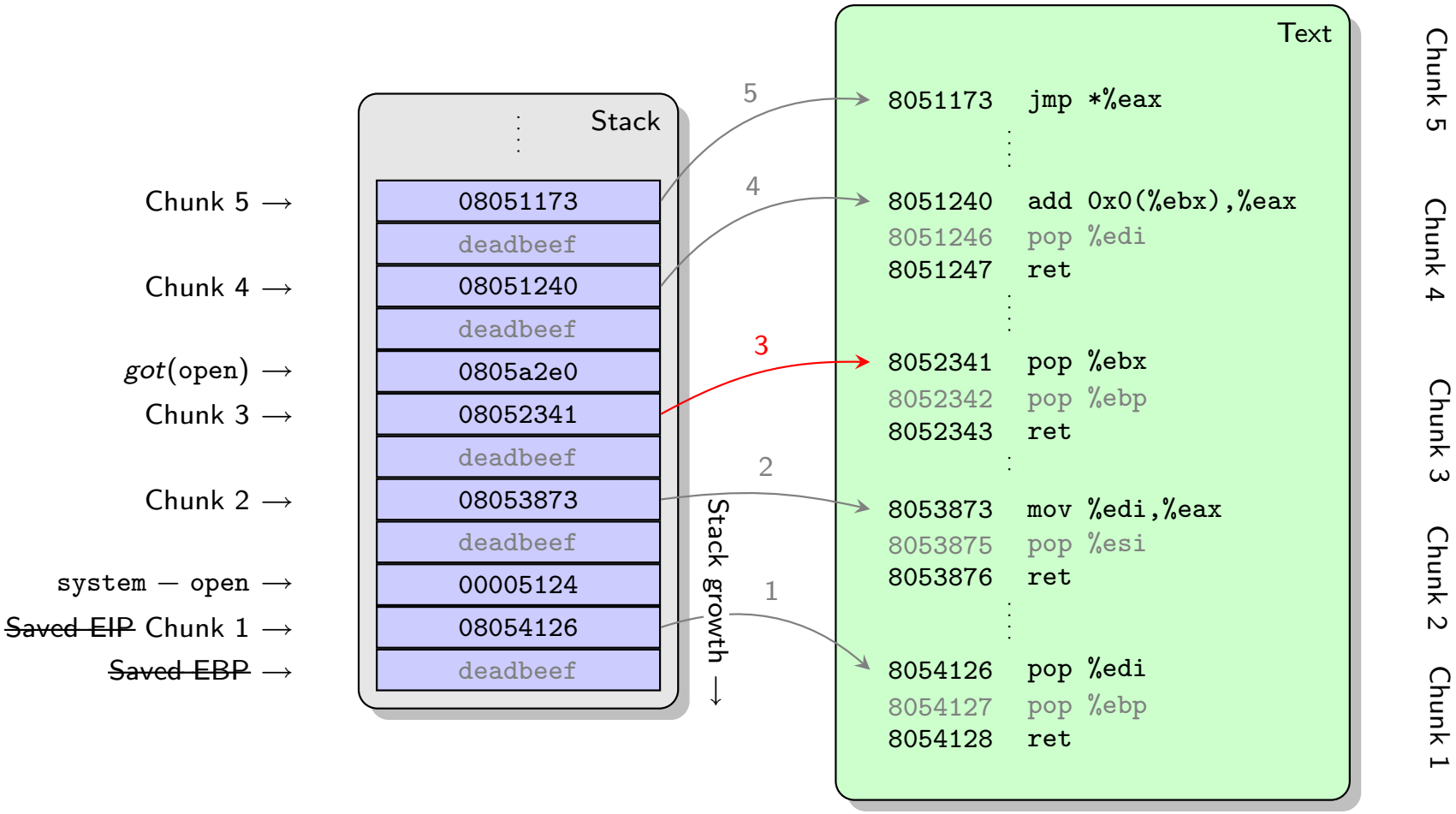
Got dereferencing



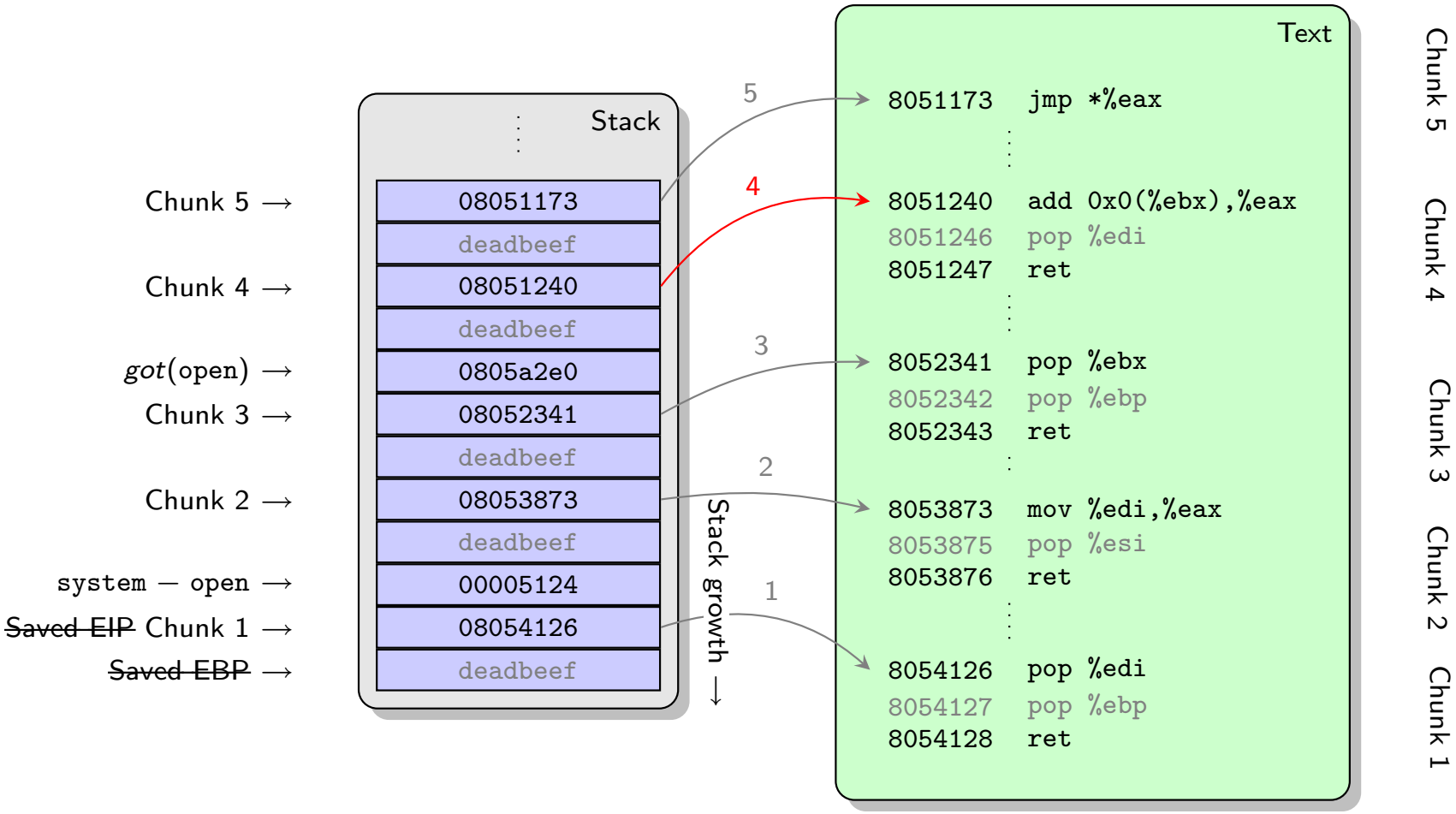
Got dereferencing



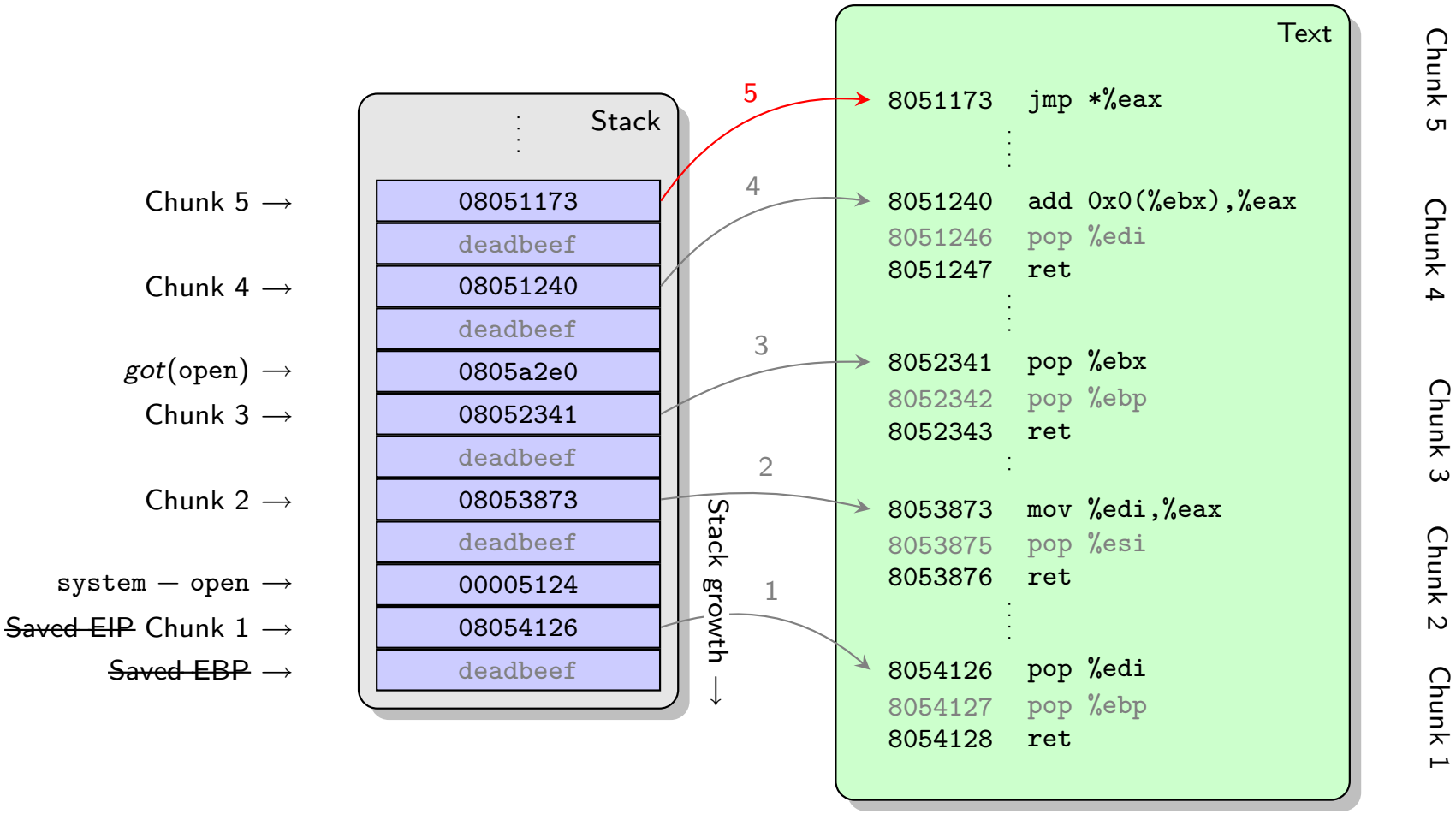
Got dereferencing



Got dereferencing



Got dereferencing



Effectiveness of our technique

	Debian (x86)	Debian (x86-64)	Fedora (x86)	OpenBSD (x86-64)
Executables	509	333	590	174
Got dereferencing	64.0%	17.8%	49.5%	58.6%
Got overwriting	96.1%	57.4%	95.0%	68.4%
Any attack	96.3%	58.3%	95.0%	68.4%

Protecting from return to randomized lib(c)

Best protection so far

Position **I**ndependent **E**xecutable (PIE)

- Complete randomization (both executable and shared libraries)
- Prevents code recycling
- Minimal overhead (1.98%)

However

- Requires recompilation
- Not widely adopted yet

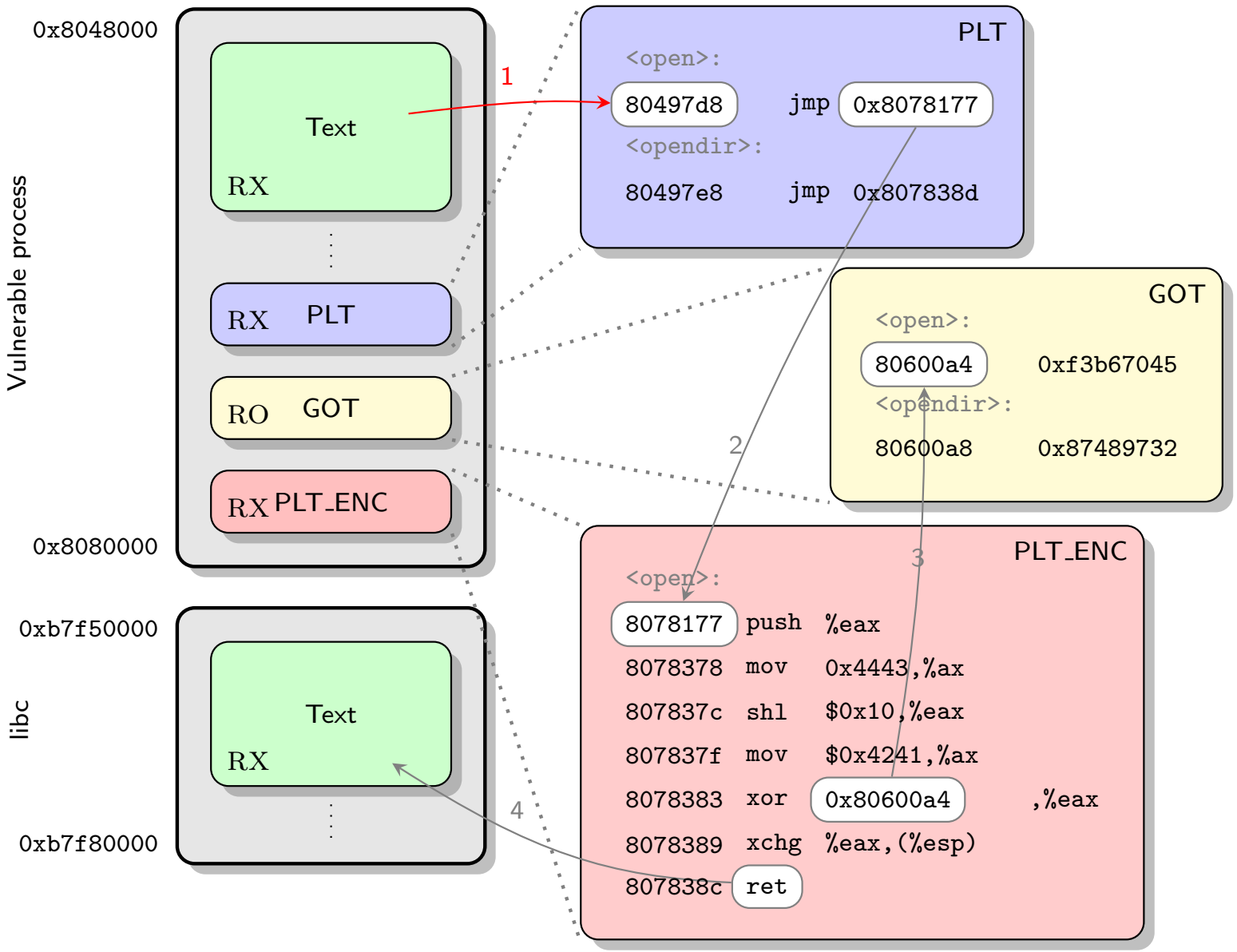
	Debian (x86)	Debian (x86-64)	Fedora (x86)	OpenBSD (x86-64)
PIE	4.3%	2.7%	14.2%	0%

Proposed protection

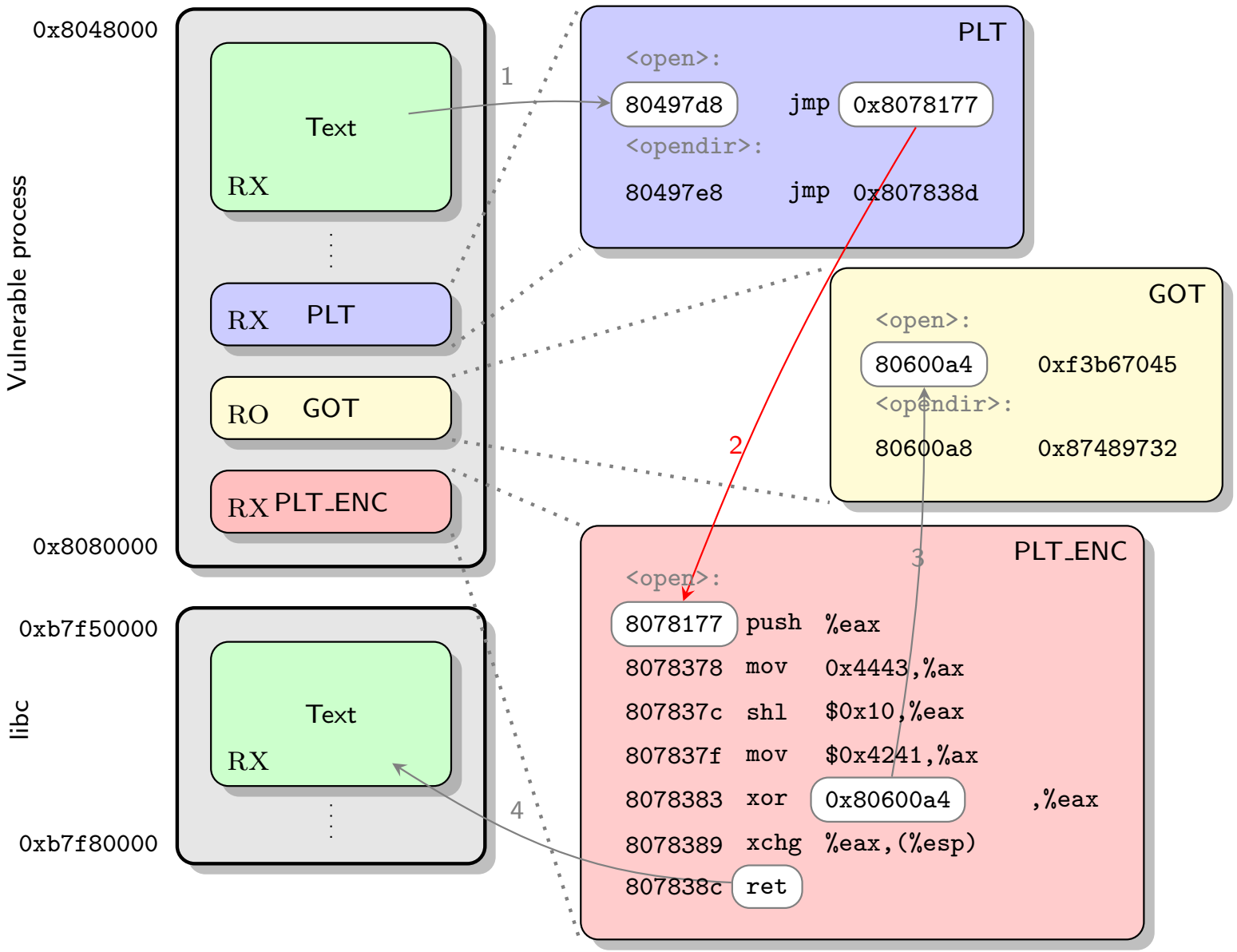
GOT encryption

- No recompilation required (preloaded library)
- Function addresses in GOT are encrypted
- `.plt` stubs patched to point to decryption routines
- Different decryption routines at every execution and for every function
- Decryption keys are built on registers
- Decryption keys and library addresses never stored in memory

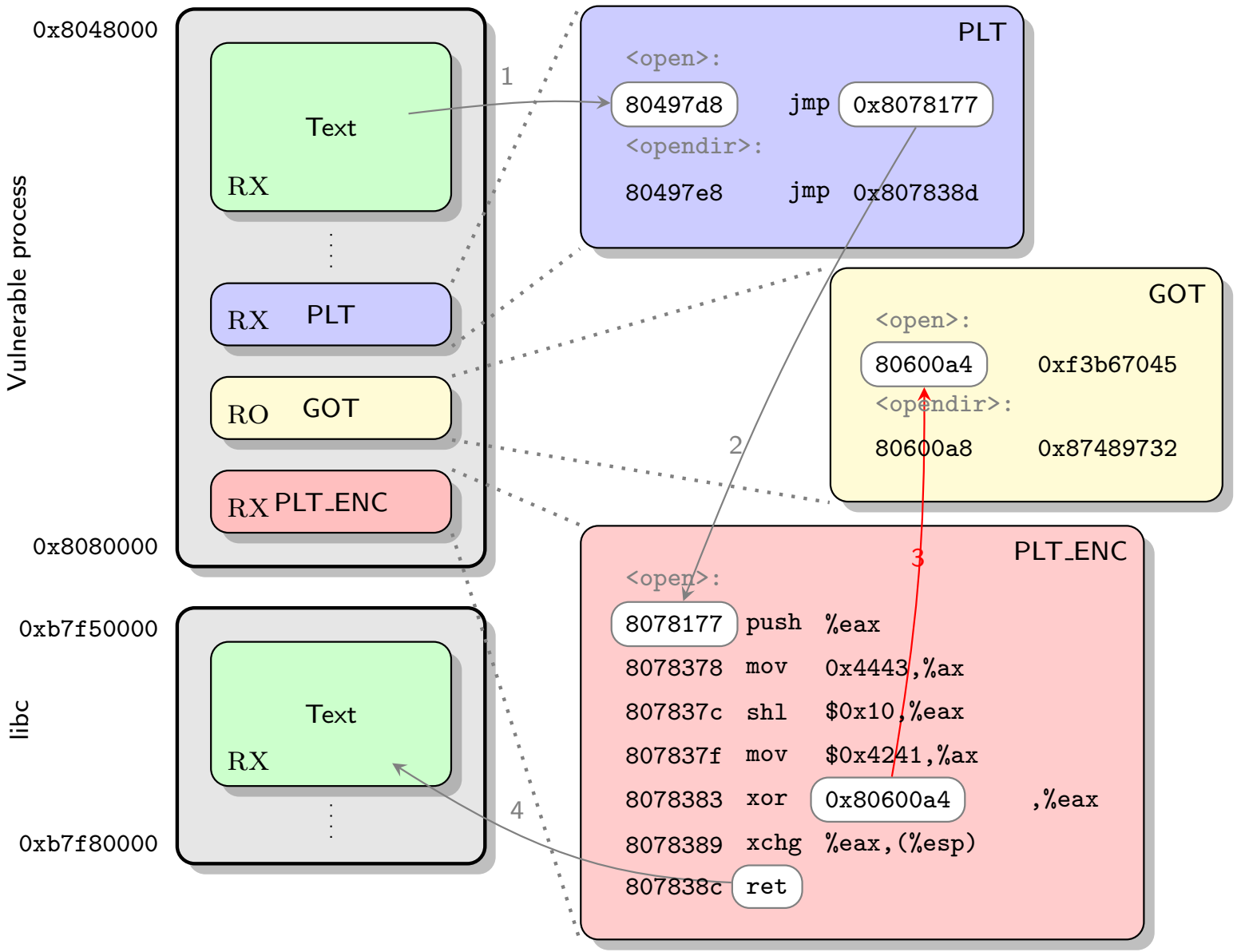
Proposed protection



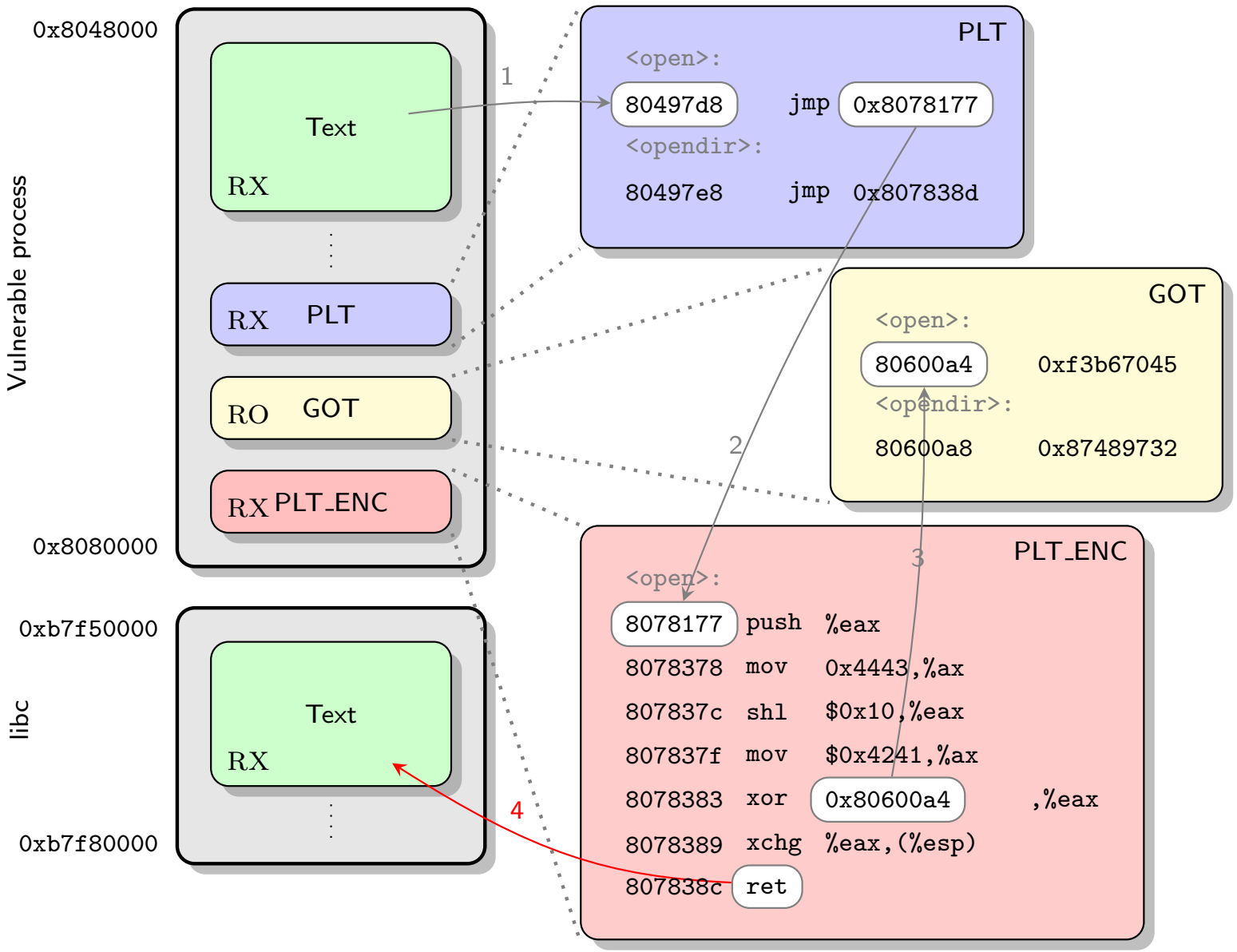
Proposed protection



Proposed protection



Proposed protection



Protection effectiveness

	PIE	Encrypted GOT
bc	10.55%	0.21%
bogofilter	3.46%	15.45%
bzip2	0%	0.63%
clamscan	0.12%	0.11%
convert	0%	0.32%
grep	1.41%	4.54%
oggenc	0.16%	0.02%
tar	0.12%	0.20%
Avg.	1.98%	2.69%

Conclusions and future works

Conclusions

- We proposed a new technique to bypass ASLR and W$\oplus$X
- We found the thechnique to be applicable to the majority of executables in most unix distributions
- We proposed a new run-time protection against our technique which has minimal overhead.

Future works

- PIE seems to be the best solution
- are PIE binaries really bullet-proof?

Surgically returning to randomized lib(c)

Thank you!
Any questions?

Giampaolo Fresi Roglia
gianz@security.dico.unimi.it

Backup slides

Got overwriting

Got overwriting

- Put an offset (system - open) into register
- Add register to memory location (open() GOT entry)
- Return to open@plt

Got overwriting