Exploiting Linux and PaX ASLR's weaknesses on 32- and 64-bit systems

Ismael Ripoll-Ripoll & Hector Marco-Gisbert
Universitat Politècnica de València (Spain)
http://cybersecurity.upv.es

Black Hat Asia 2016, Singapore

March 29 - April 1, 2016

Table of contents

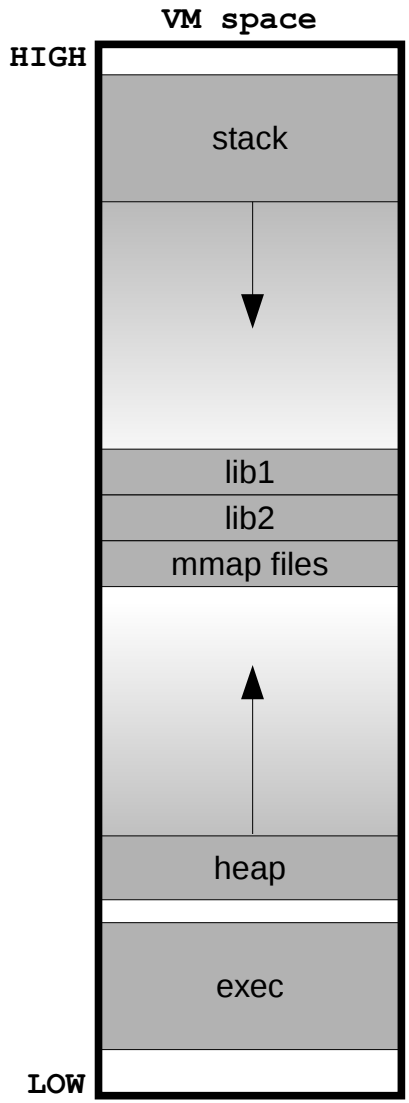
1. ASLR overview & background
2. Linux and PaX ASLR weaknesses
   - Too low entropy
   - Non-uniform distribution
   - Correlation between objects
   - Inheritance
3. ASLRA: ASLR Analyzer
4. Exploiting the Correlation weakness: **offset2lib**
   - Example: Offset2lib in stack buffer overflows
   - Demo: Root shell in < 1 sec.
   - Mitigations
5. ASLR-NG: ASLR Next Generation
   - New randomisation forms
   - Linux vs PaX vs ASLR-NG
6. Conclusions

ASLR background

What is ASLR ?

- ASLR does not remove vulnerabilities but makes it more difficult to exploit them.
- ASLR deters exploits which relays on knowing the memory map (or a part of it).
- ASLR is effective when all memory areas are randomized. Otherwise, the attacker can use these non-random areas.
- Full ASLR is when:
  - Applications are compiled with PIE (-fpie -pie).
  - And randomize_va_space = 2

  (all is randomized. stack, VDSO, shared memory, data segment)



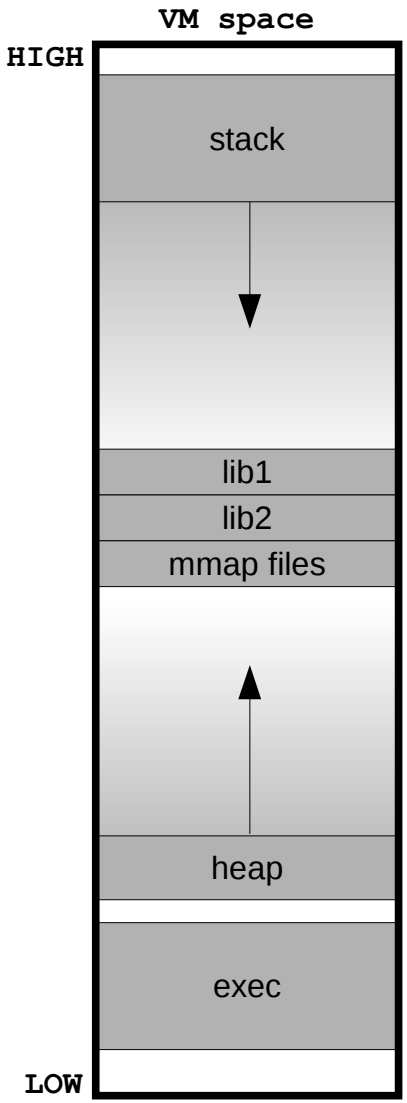
Randomized objects

There are several things (objects) that can be randomized:

1. The Arguments (and environmental vars) passed to the process.
2. The Main Stack.
3. The Heap.
4. The executable.
5. The loader (ld-)
6. The libraries.
7. vDSO
8. The maps allocated with the `mmap()` call:
   - Huge pages, memory objects that use large pagetable entries.
   - The Stack of the threads.
   - ...

How the ASLR works: A simple example

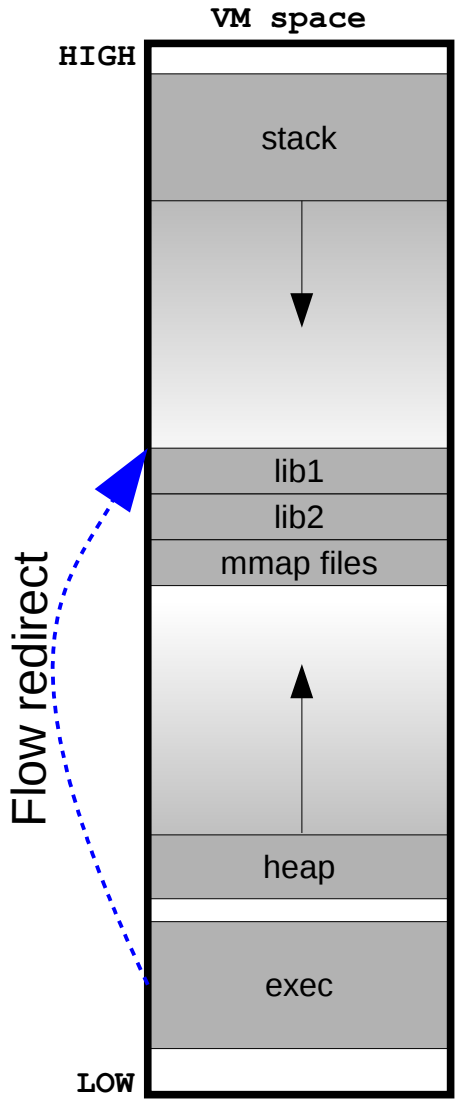
The attacker redirects the execution to the `exec()` library function (`lib1`):



How the ASLR works: A simple example

The attacker redirects the execution to the `exec()` library function (`lib1`):

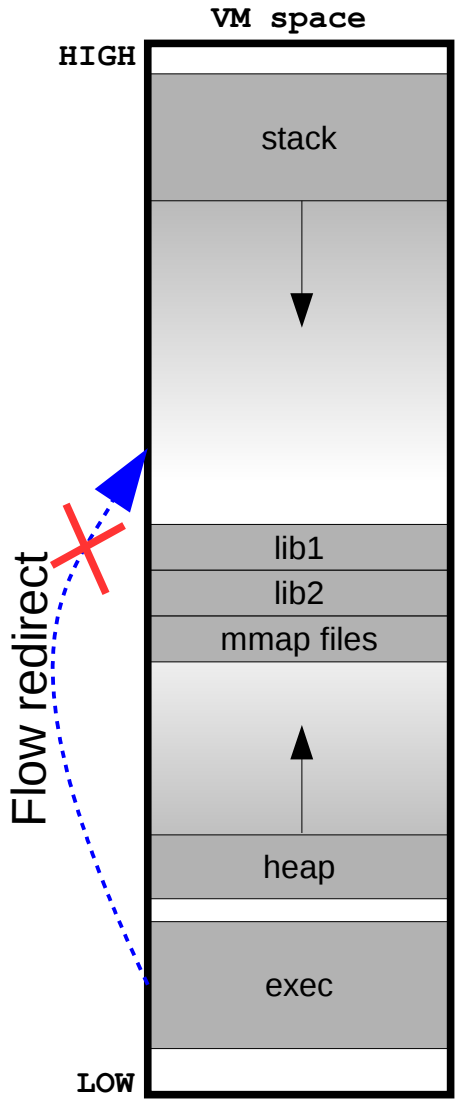
- Trivial if ASLR is off.



How the ASLR works: A simple example

The attacker redirects the execution to the `exec()` library function (`lib1`):

- Trivial if ASLR is off.
- But it fails when the ASLR is on.



How the ASLR works: A simple example

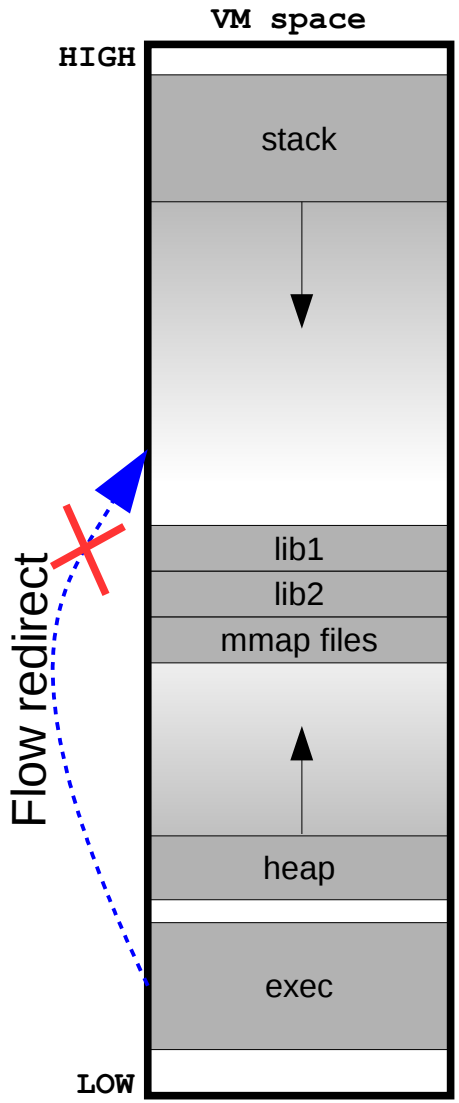
The attacker redirects the execution to the `exec()` library function (`lib1`):

- Trivial if ASLR is off.
- But it fails when the ASLR is on.

Not only the libraries but all other memory areas are randomized:

→ How difficult is to predict the target ?
- Depends on the entropy.

→ Are there other attack vectors ?
- Yes, We have found new weaknesses.



Linux and PaX ASLR weaknesses

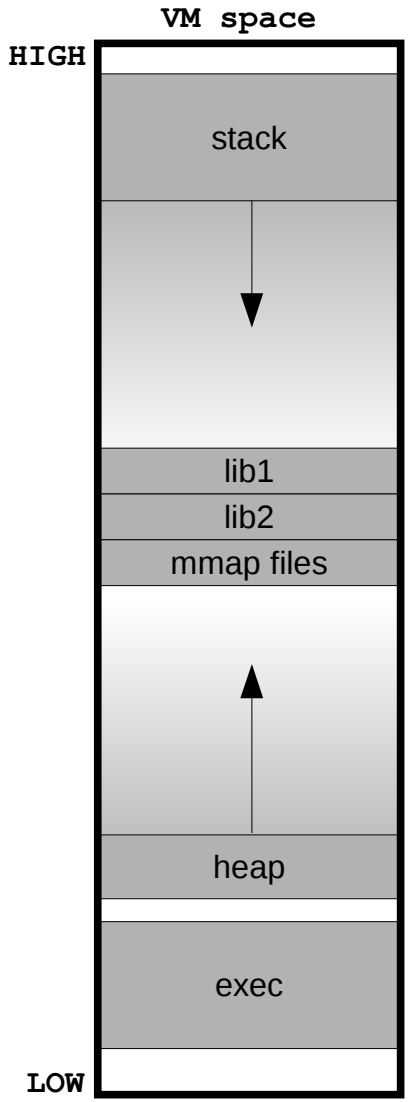
Linux and PaX ASLR weaknesses

Current ASLR was designed considering that some zones are growable but:

- Cannot be used safely because collisions with other allocations cannot be avoided.
- Currently, only used in the Stack and the Heap.

Growable objects impose strong limitations on ASLR design:

- Linux places each object as separately as possible (Stack and Heap)
- Unfortunately, this **introduces weaknesses**.

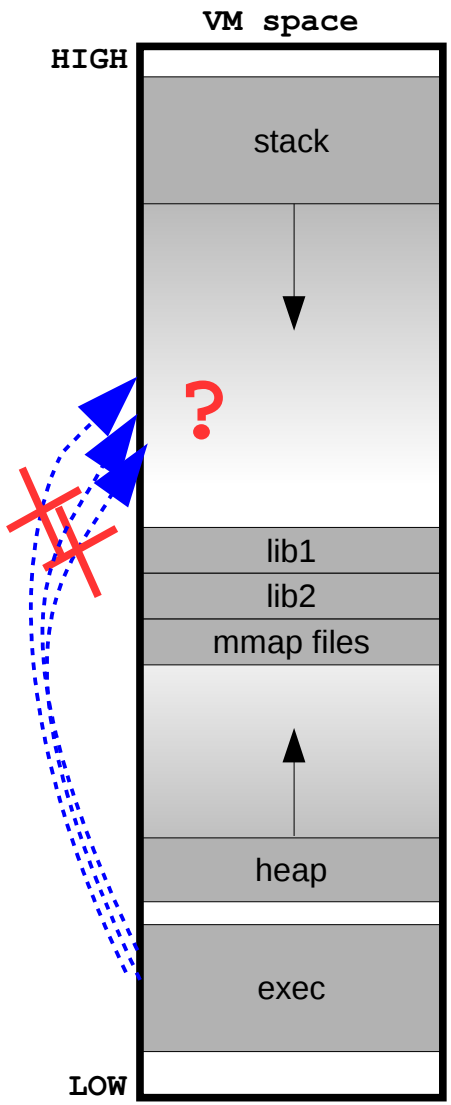


Weakness 1) Too low entropy

Brute force attacks to bypass ASLR:

	x86	Entropy	100%	Mean
32-bit	Linux	2^{8}	< 1 Sec	< 1 Sec
	PaX	~~2^{16}~~ * 2^{15}	32 Sec	16 Sec
64-bit	Linux	2^{28}	74 Hours	37 Hours
	PaX	~~2^{28}~~ * 2^{27}	37 Hours	18 Hours

Entropy and time (1000 trials/sec, **EXEC** object).

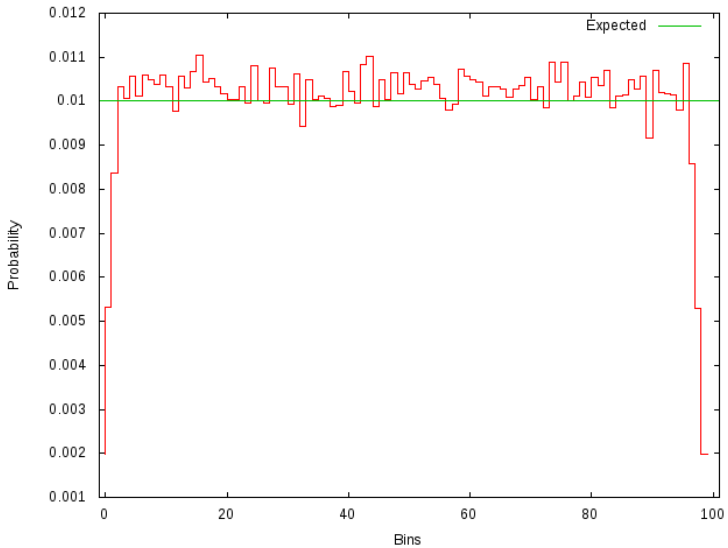


- In 32-bit very low entropy.
- In 64-bit the attack is feasible in some scenarios.

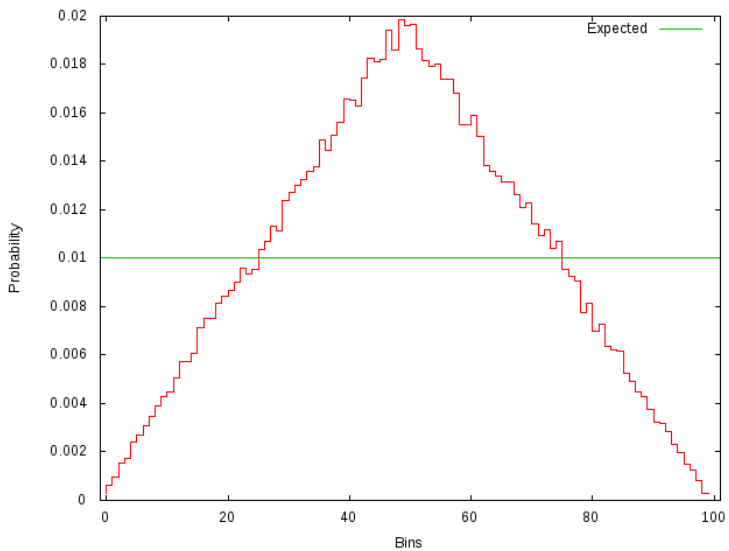
* Paxtest tool measures these values incorrectly. It measures the entropy of the library rather than the executable.

Weakness 2) Non-uniform distribution

- A non-uniform distribution is not automatically a bad thing.
- But it is bad when the attacker can design faster attacks by focusing on the most frequent (likely) addresses.
- Entropy estimation is hard (the paxtest is not precise).

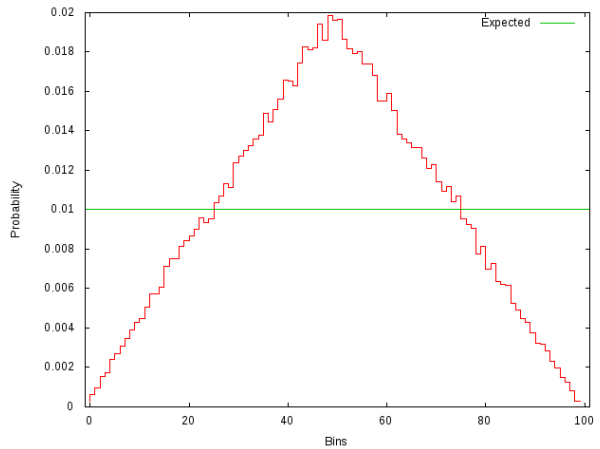


There is a large and flat area to attack.

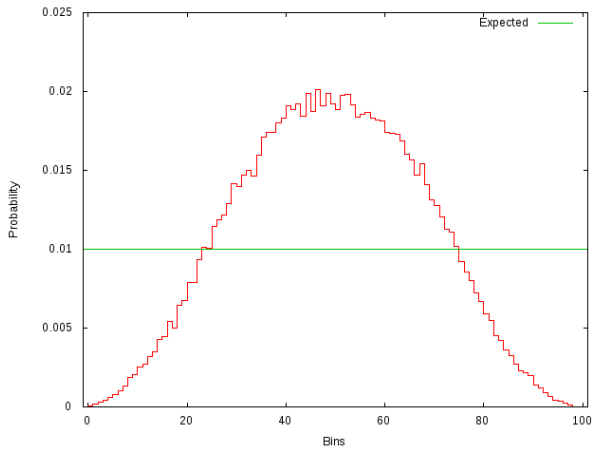


We can focus the attack on a few values.

Weakness 2) Non-uniform distribution



PaX Libraries distribution in 32-bit



PaX Libraries distribution in 64-bit

Libraries are not uniformly distributed:

- The triangular distribution (left side) is caused by the sum of two random variables of the same range: $addr = rnd(2^{15}) + rnd(2^{15})$;
- The Irwin-Hall distribution (right side) is the result of summing three random vars: $addr = rnd(2^{27}) + rnd(2^{27}) + rnd(2^{27})$;

Weakness 3) Correlation between objects

Instead of de-randomizing the target object, first de-randomize an intermediate (accesible or leakable) object and then use it to de-randomize the target object.

We developed a PoC exploit (**Offset2lib**) to show the libraries $\leftrightarrow$ executable correlation in Linux:

- It bypass the full Linux ASLR on any architecture in < 1 sec.
- It does not use the *GOT* or *PLT*.
- It works even with NX and SSP enabled.
- It exploits a stack buffer overflow.

The **Metaphor**[1] (March 2016) exploit relies on the correlation property of the libraries on Android (and also on the low entropy of Linux in 32-bits).

[1] http://north-bit.com

Weakness 3) Correlation between objects

- The correlation is specially interesting when:
  - Two objects, A and B, are correlated.
  - Object B has more entropy than A.
  - The conditional entropy (entropy of B once you know A) is less than the absolute entropy B.
- This is the case when using HUGE PAGES in Linux ($\geq$ 3.12) 64-bit.
  - Hugepages: 19 bits of entropy.
  - Libraries: 28 bits of entropy.
  - Co-entropy: 9 bits of entropy.
  - Brute-force Libraries: 2^{28} (3 days).
  - Brute-force Hugepage then Library: $2^{19} + 2^{9}$ (8 minutes)
  - This case is also present in PaX.

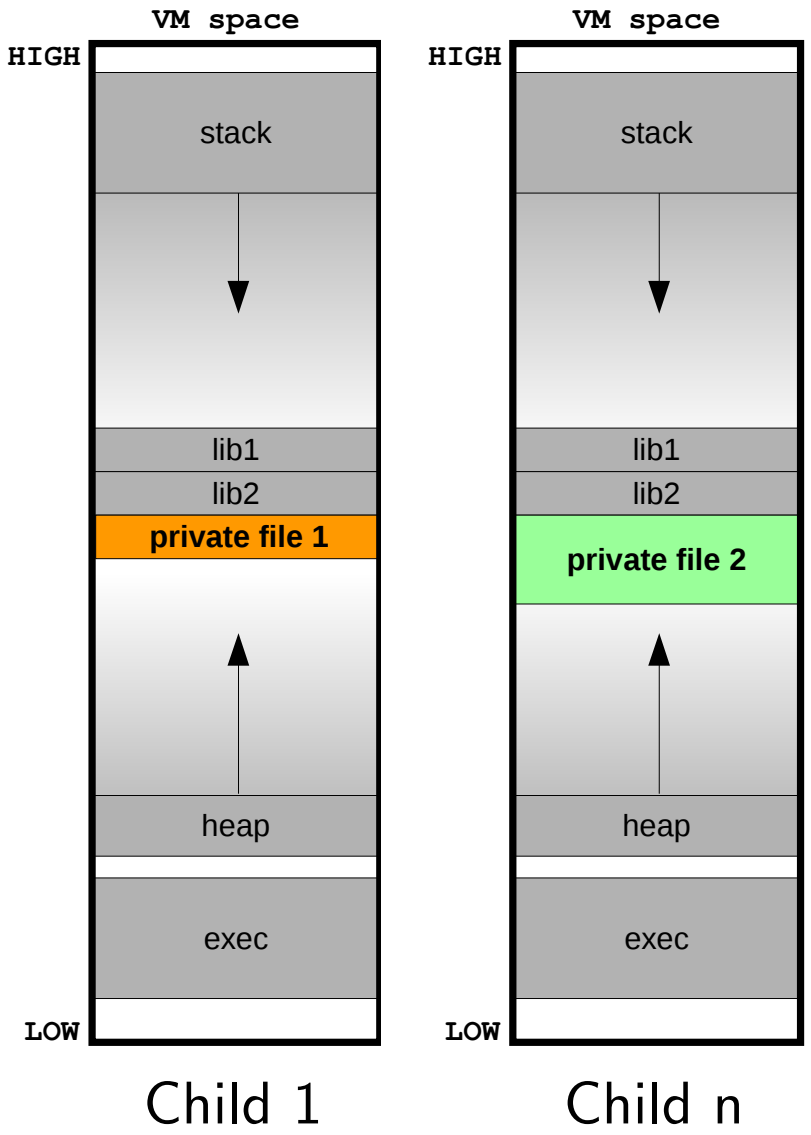
Huge pages are typically used in databases. Assuming 1000 trials x sec.

Weakness 4) Inheritance

There is not entropy between the memory map of the father and his children (`fork()`).

All forked processes share the same memory layout.

And so the attackers can predict where new objects will be in memory for its sibling.



Weakness 4) Inheritance

- This affects specially applications/architectures that use the forking model like: Android with Zygote, KDEinit, Chrome browser, etc.
- For example, an info leak of an Android application can be used to de-randomize all libraries and applications running in the same system (until the system is rebooted).
- In those cases, the attacker can even predict the layout of future processes, once the layout of any of the children or father is known.

The SSH suite uses the `fork()+exec()` sequence to renew all the secrets (included ASLR) before new connections.

ASLRA: ASLR Analyzer.

ASLRA: ASLR Analyzer

- Unfortunately, the `paxtest` tool is not powerful enough to make and rigorous study of the ASLR.
- We have had to developed ASLRA, a test suit to deeply analyze the entropy of objects. ASLRA is composed of two tools:
  1. Sampler:
     → An application which generates millions of samples (address of mapped objects) and saves the raw data.
  2. Analyzer:
     → Computes several statistical parameters.
     → Individual byte, Shannon entropy, flipping bits, etc.

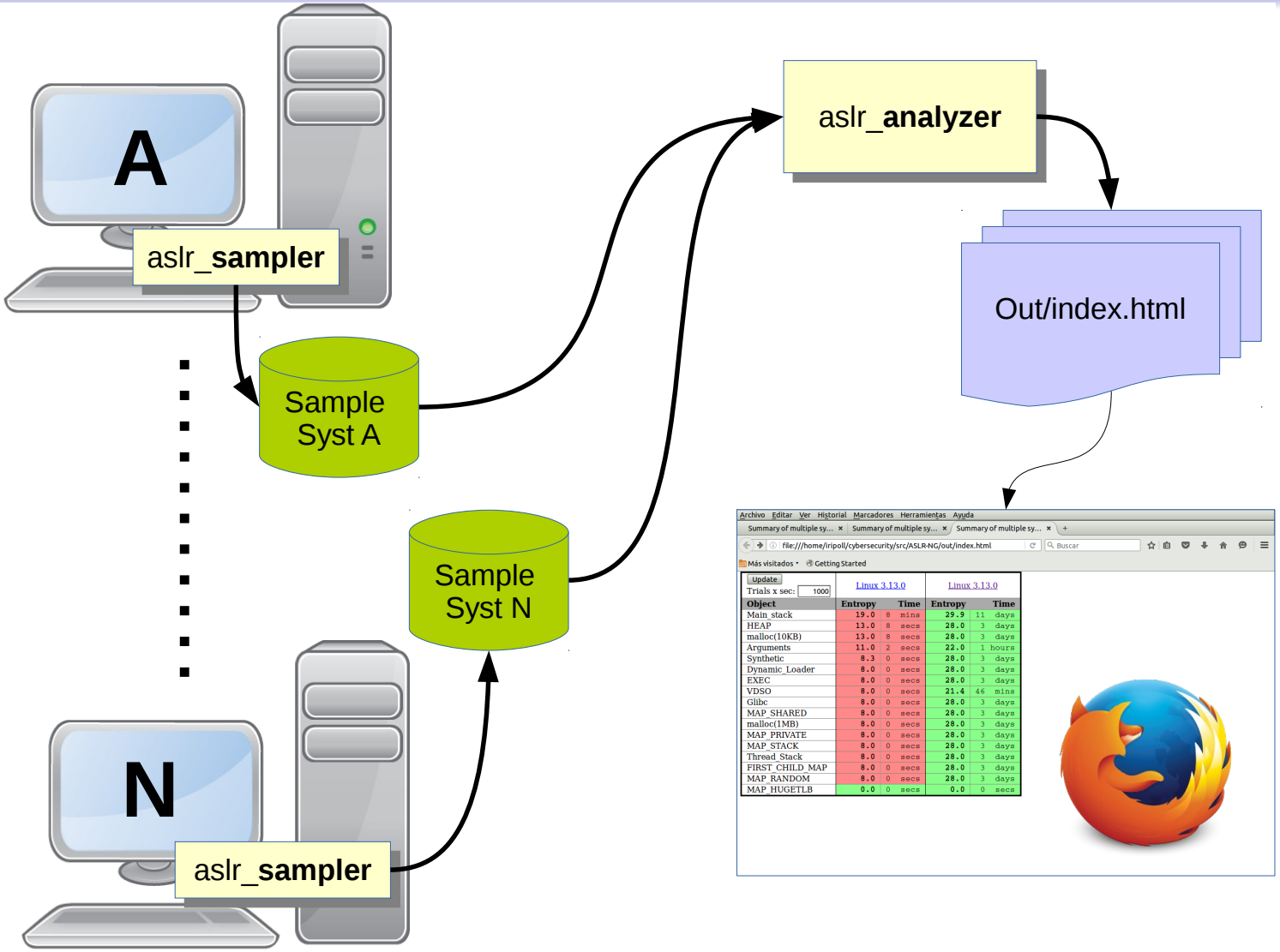
ASLRA: ASLR Analyzer

- Unfortunately, the `paxtest` tool is not powerful enough to make and rigorous study of the ASLR.
- We have had to developed ASLRA, a test suit to deeply analyze the entropy of objects. ASLRA is composed of two tools:
  1. Sampler:
     → An application which generates millions of samples (address of mapped objects) and saves the raw data.
  2. Analyzer:
     → Performs the statistical analysis.
     → Distribution plots.
     → Individual byte, Shannon entropy, flipping bits, etc.

$$\downarrow$$

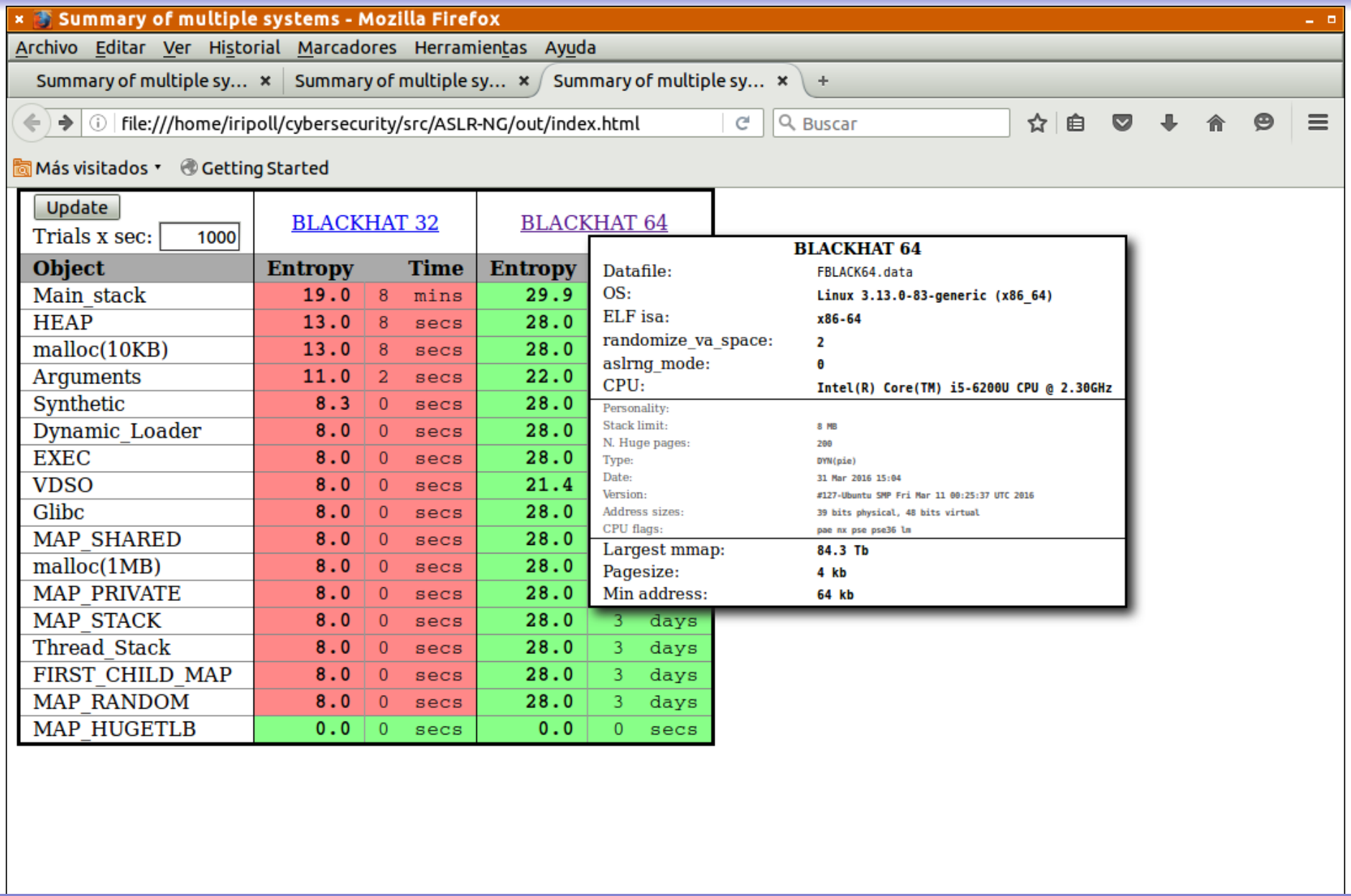
$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x)$$

ASLR-NG: ASLR analyzer tool



ASLRA data flow example.

ASLR-NG: ASLR analyzer tool



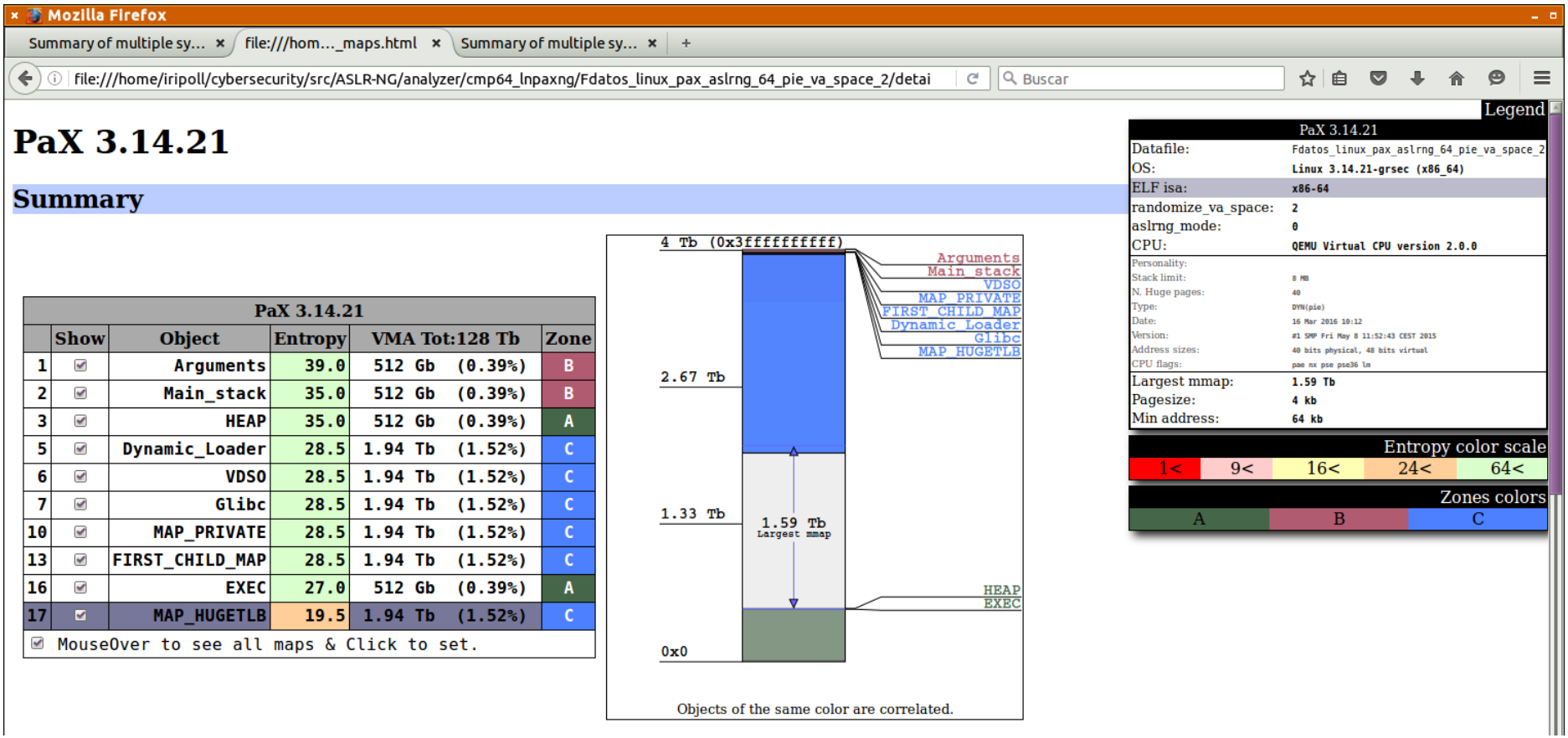
Update
Trials x sec: 1000

Object	BLACKHAT 32 Entropy	Time	BLACKHAT 64 Entropy	Time
Main_stack	19.0	8 mins	29.9	1 days
HEAP	13.0	8 secs	28.0	3 days
malloc(10KB)	13.0	8 secs	28.0	3 days
Arguments	11.0	2 secs	22.0	3 days
Synthetic	8.3	0 secs	28.0	3 days
Dynamic_Loader	8.0	0 secs	28.0	3 days
EXEC	8.0	0 secs	28.0	3 days
VDSO	8.0	0 secs	21.4	0 secs
Glibc	8.0	0 secs	28.0	3 days
MAP_SHARED	8.0	0 secs	28.0	3 days
malloc(1MB)	8.0	0 secs	28.0	3 days
MAP_PRIVATE	8.0	0 secs	28.0	3 days
MAP_STACK	8.0	0 secs	28.0	3 days
Thread_Stack	8.0	0 secs	28.0	3 days
FIRST_CHILD_MAP	8.0	0 secs	28.0	3 days
MAP_RANDOM	8.0	0 secs	28.0	3 days
MAP_HUGETLB	0.0	0 secs	0.0	0 secs

BLACKHAT 64

Datafile:	FBLACK64.data
OS:	Linux 3.13.0-83-generic (x86_64)
ELF isa:	x86-64
randomize_va_space:	2
aslrng_mode:	0
CPU:	Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
Personality:	
Stack limit:	8 MB
N. Huge pages:	200
Type:	DYN(pie)
Date:	31 Mar 2016 15:04
Version:	#127-Ubuntu SMP Fri Mar 11 00:25:37 UTC 2016
Address sizes:	39 bits physical, 48 bits virtual
CPU flags:	pae nx pse pse36 lm
Largest mmap:	84.3 Tb
Pagesize:	4 kb
Min address:	64 kb

ASLR-NG: ASLR analyzer tool



Screen shot of the summary of a system.

ASLR-NG: ASLR analyzer tool

Detailed object analysis

Arguments | Main_stack | HEAP | Dynamic_Loader | VDSO | Glibc | MAP_PRIVATE | FIRST_CHILD_MAP | EXEC | MAP_HUGETLB

MAP_HUGETLB

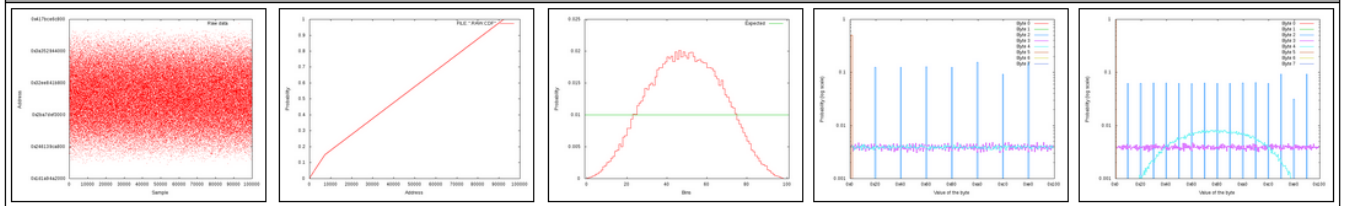
Basic statistics (I)	
Samples:	100000
Errors:	0
Range:	0x1f1a8200001
Min:	0x205d1800000
Max:	0x3f779a00000
Random bits:	**20**
Shannon entropy:	**19.52**
Sum of byte entropy:	**19.97**
And-ed:	0x20000000000
Or-ed:	0x3ffffe00000
Bits that change:	0x1ffffe00000
Low unchanged bits:	21

Basic statistics (II)	
Mean measured:	0x2ffa063861a
Mean theoric:	0x2fea5900000
Mean error:	0.13%
Mode:	0x36489200000
Mode repetitions:	5
Most likely:	0x2f42e000000
Median measured:	0x2ff90a00000
Median theoric:	0x2fea5900000
Median error:	0.12%
StdDev measured:	0x5a141fd740
StdDev theoric:	0x8fa93f8c1e
StdDev error:	37.30%

Chi2 goodness fit to uniform			
nº bins	ref. low	val	ref. high
32	15.65	49547.70	52.19
64	39.85	49719.48	92.01
100	69.23	49787.53	134.64
256	205.42	49929.96	310.46
1024	920.72	50736.68	1131.16
2048	1901.10	51764.54	2198.78
4096	3887.41	53692.49	4308.47
8192	7896.19	57653.63	8491.69
8263	7965.90	57474.75	8563.98

Chi2 test 99% confidence of being a uniform.

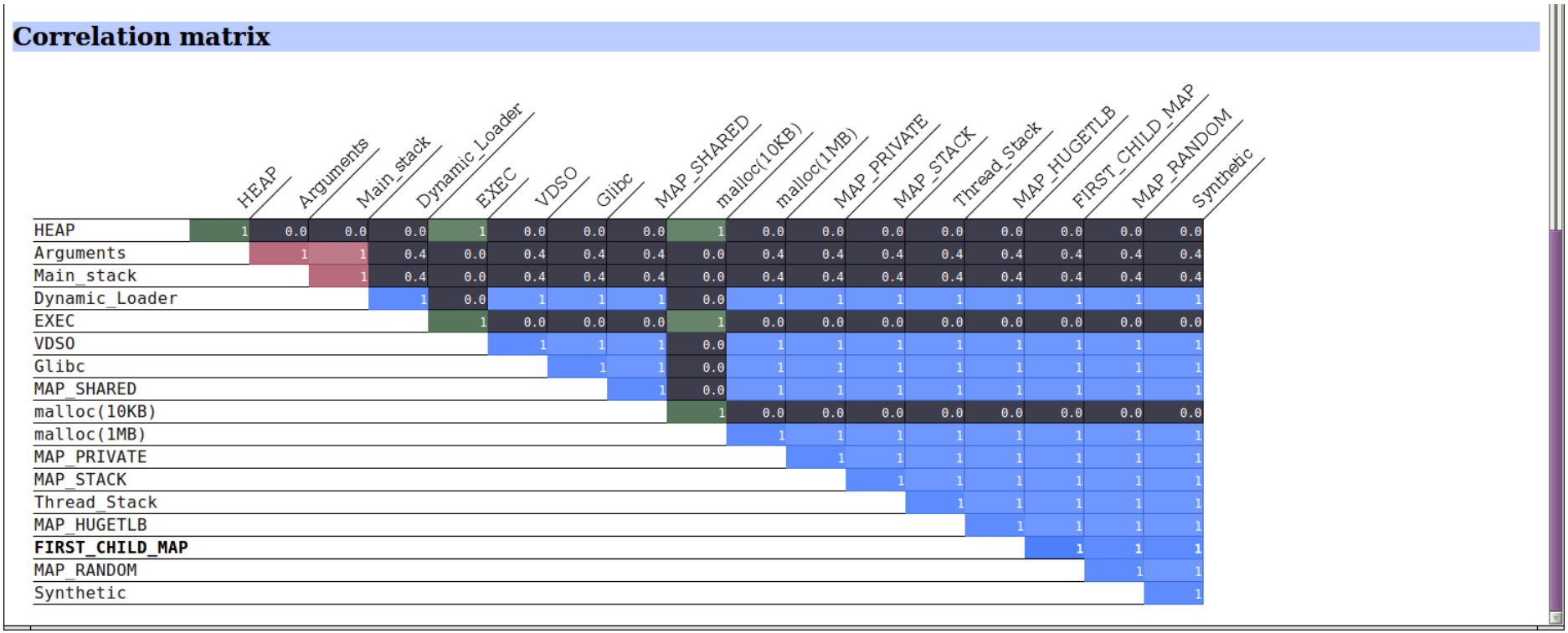
Distributions



Raw data plot. Cumulative Distribution Func. (CDF). Histogram (100 bins). Individual byte histogram. Altered individual byte histogram.

Detailed view of an object.

ASLR-NG: ASLR analyzer tool



Correlation matrix

	HEAP	Arguments	Main_stack	Dynamic_Loader	EXEC	VDSO	Glibc	MAP_SHARED	malloc(10KB)	malloc(1MB)	MAP_PRIVATE	MAP_STACK	Thread_Stack	MAP_HUGETLB	FIRST_CHILD_MAP	MAP_RANDOM	Synthetic
HEAP	1	0.0	0.0	0.0	1	0.0	0.0	0.0	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Arguments		1	1	0.4	0.0	0.4	0.4	0.4	0.0	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
Main_stack			1	0.4	0.0	0.4	0.4	0.4	0.0	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
Dynamic_Loader				1	0.0	1	1	1	0.0	1	1	1	1	1	1	1	1
EXEC					1	0.0	0.0	0.0	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
VDSO						1	1	1	0.0	1	1	1	1	1	1	1	1
Glibc							1	1	0.0	1	1	1	1	1	1	1	1
MAP_SHARED								1	0.0	1	1	1	1	1	1	1	1
malloc(10KB)									1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
malloc(1MB)										1	1	1	1	1	1	1	1
MAP_PRIVATE											1	1	1	1	1	1	1
MAP_STACK												1	1	1	1	1	1
Thread_Stack													1	1	1	1	1
MAP_HUGETLB														1	1	1	1
FIRST_CHILD_MAP															1	1	1
MAP_RANDOM																1	1
Synthetic																	1

Correlation matrix.

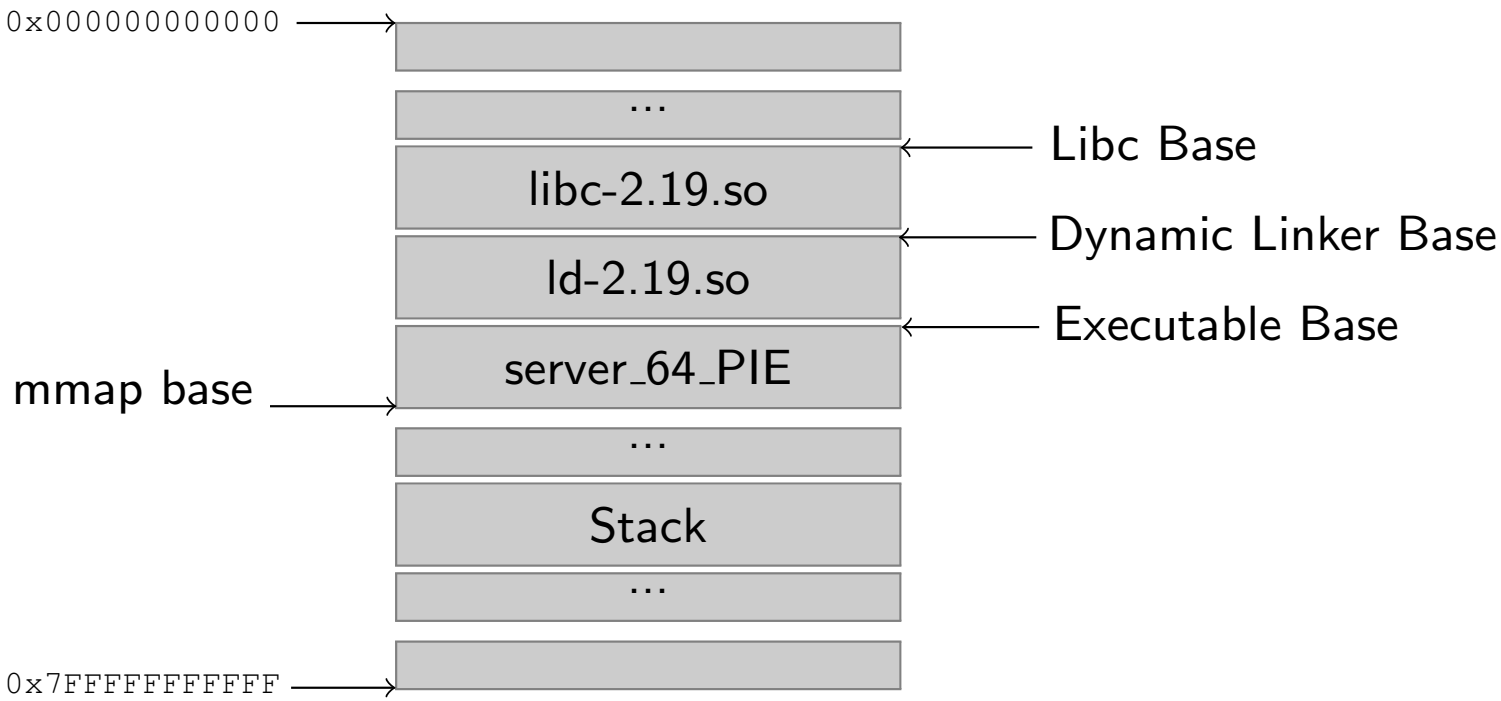
Exploiting the correlation weaknesses

Offset2lib example: Bypassing NX, SSP and ASLR in x86_64 $<$ 1 sec

Offset2lib: Loading shared objects

The problem appears when the application is compiled with PIE because the GNU/Linux algorithm for loading shared objects works as follows:

- The **first** shared object is loaded at a **random position**.
- The next object is located right below (lower addresses) the last object.



All libraries are located "side by side" at a **single random place**.

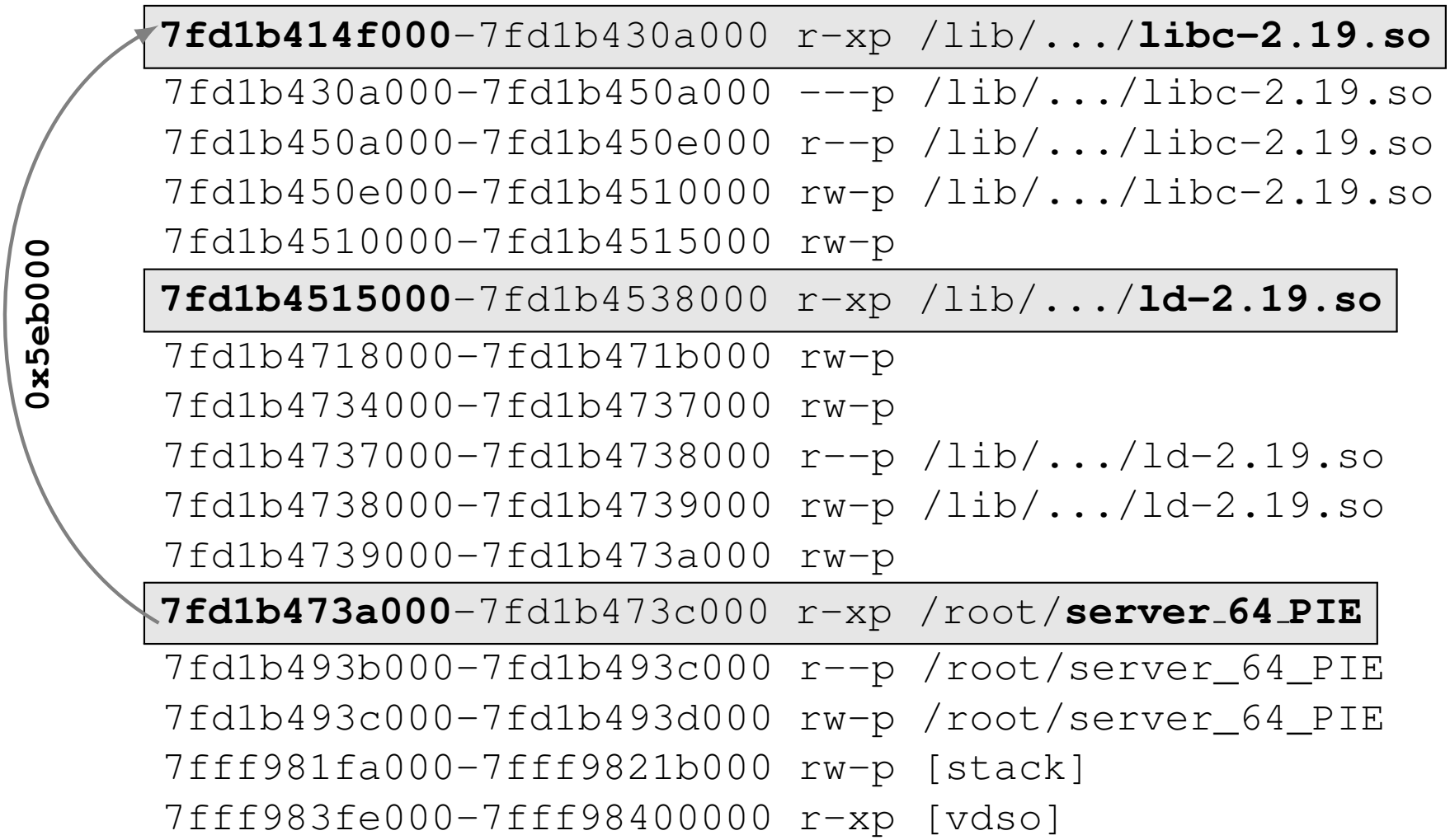
Offset2lib

$ cat /proc/<pid>/server_64_PIE

```
7fd1b414f000-7fd1b430a000 r-xp /lib/.../libc-2.19.so
7fd1b430a000-7fd1b450a000 ---p /lib/.../libc-2.19.so
7fd1b450a000-7fd1b450e000 r--p /lib/.../libc-2.19.so
7fd1b450e000-7fd1b4510000 rw-p /lib/.../libc-2.19.so
7fd1b4510000-7fd1b4515000 rw-p
7fd1b4515000-7fd1b4538000 r-xp /lib/.../ld-2.19.so
7fd1b4718000-7fd1b471b000 rw-p
7fd1b4734000-7fd1b4737000 rw-p
7fd1b4737000-7fd1b4738000 r--p /lib/.../ld-2.19.so
7fd1b4738000-7fd1b4739000 rw-p /lib/.../ld-2.19.so
7fd1b4739000-7fd1b473a000 rw-p
7fd1b473a000-7fd1b473c000 r-xp /root/server_64_PIE
7fd1b493b000-7fd1b493c000 r--p /root/server_64_PIE
7fd1b493c000-7fd1b493d000 rw-p /root/server_64_PIE
7fff981fa000-7fff9821b000 rw-p [stack]
7fff983fe000-7fff98400000 r-xp [vdso]
```

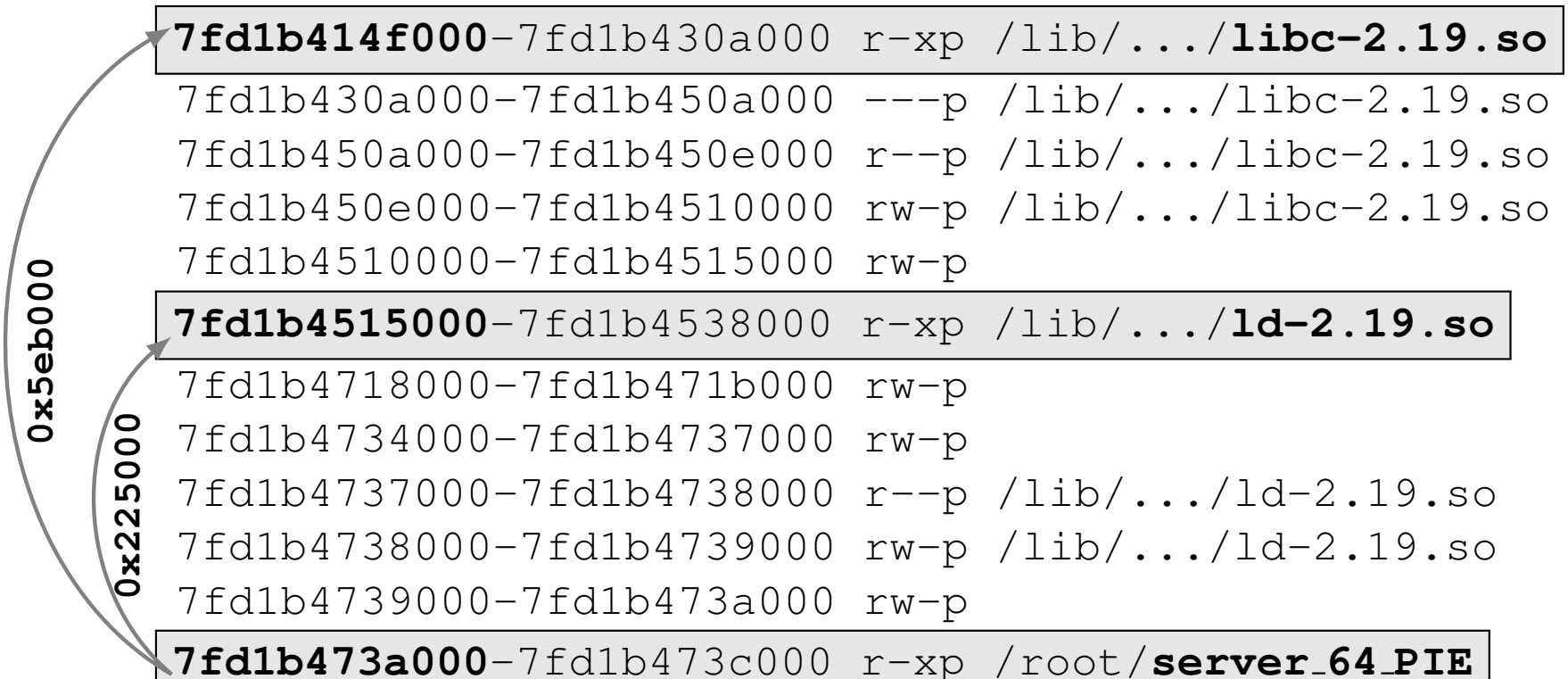
Offset2lib

$ cat /proc/<pid>/server_64_PIE

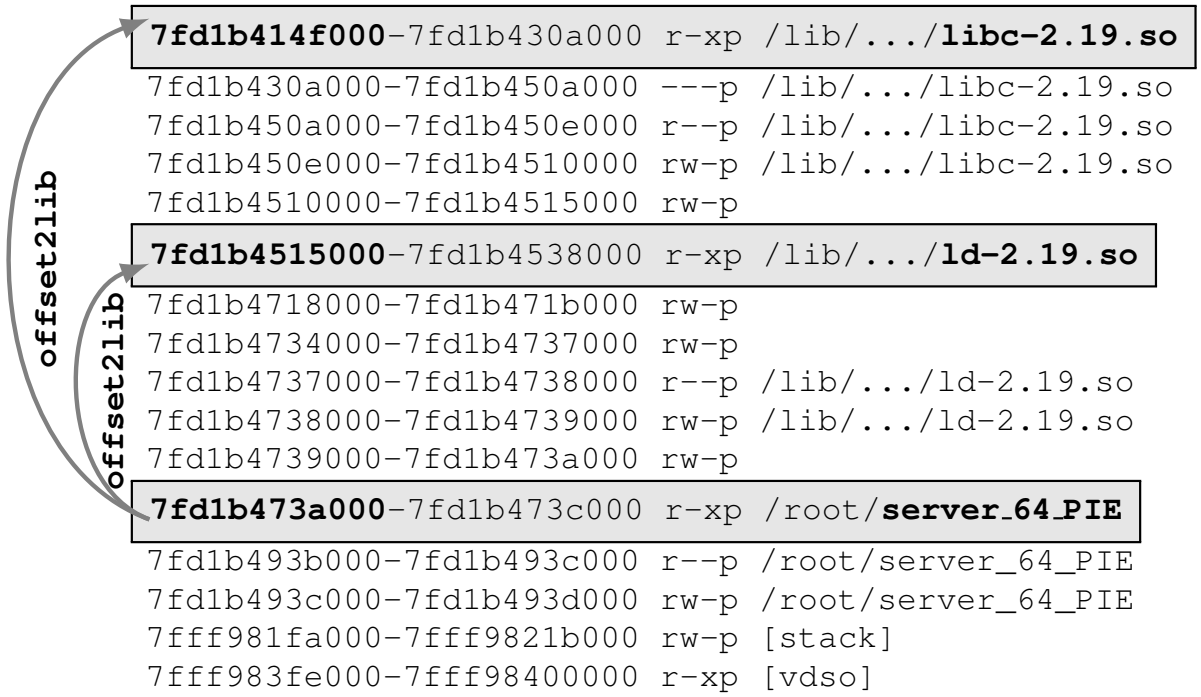


Offset2lib

$ cat /proc/<pid>/server_64_PIE



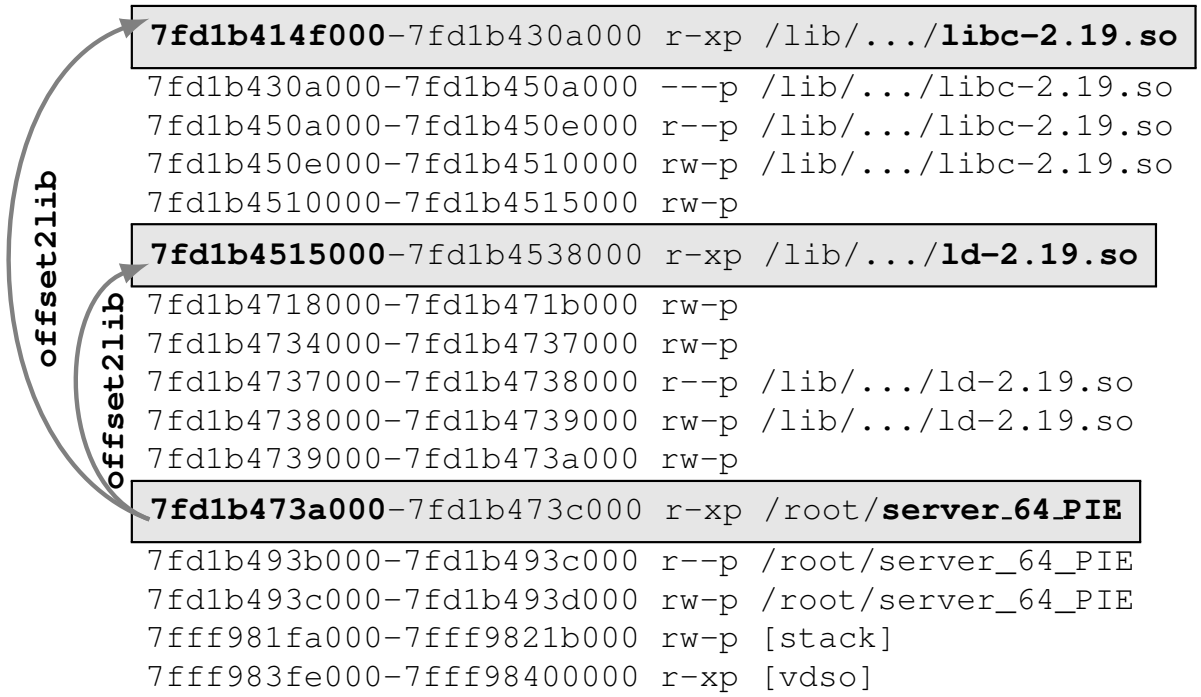
Offset2lib



We named this invariant distance **offset2lib** which:

- It is a **constant distance** between two shared objects even in different executions of the application.

Offset2lib



We named this invariant distance **offset2lib** which:

- It is a **constant distance** between two shared objects even in different executions of the application.

Any address of the app. → de-randomize all mmapped areas !!!

Why the Offset2lib is dangerous ?

Offset2lib scope:

- **Realistic**; applications are more prone than libraries to errors.
- Makes some vulnerabilities **faster**, **easier** and **more reliable** to exploit them.
- It is not a self-exploitable vulnerability but an ASLR-design weakness exploitable.
- It opens new (and old) attack vectors.

Why the Offset2lib is dangerous ?

Offset2lib scope:

- **Realistic**; applications are more prone than libraries to errors.
- Makes some vulnerabilities **faster**, **easier** and **more reliable** to exploit them.
- It is not a self-exploitable vulnerability but an ASLR-design weakness exploitable.
- It opens new (and old) attack vectors.

Next example:
Offset2lib on a standard stack buffer overflow.

Offset2lib in stack buffer overflows

The steps to build the attack are:

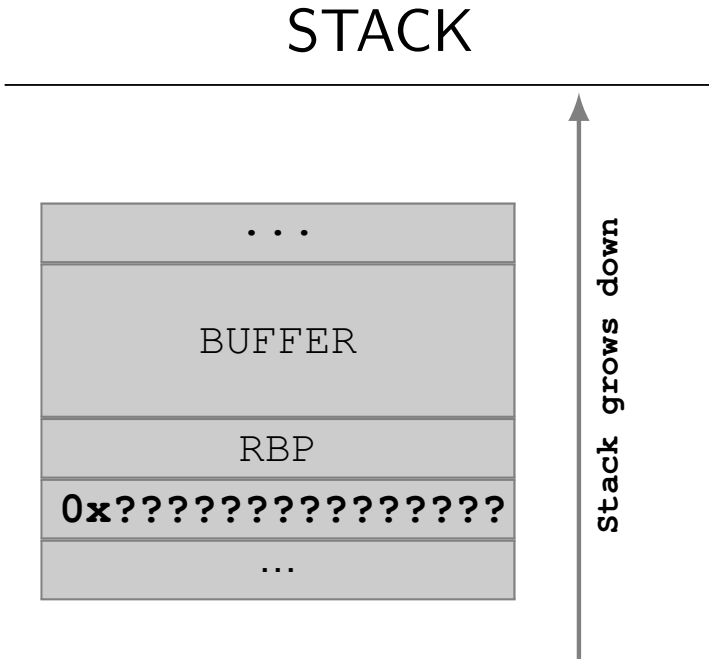
1. Extracting static information
2. Brute force part of saved-IP
3. Calculate base app. address
4. Calculate library offsets
5. Obtain mmapped areas

1) Extracting static information

→ Our goal is to obtain an address belonging to the application.
→ We are going to obtain the **saved-IP** of vulnerable function caller.

Offset2lib with **saved-IP** ⇒ **all** mmapped areas.

```
0000000000001063 <attend_client>:
1063: 55                    push %rbp
1064: 48 89 e5              mov  %rsp,%rbp
1067: 48 81 ec 60 04 00 00  sub  $0x460,%rsp
106e: 64 48 8b 04 25 28 00  mov  %fs:0x28,%rax
1075: 00 00
..... .....         .....
12d7: 48 89 c7              mov  %rax,%rdi
12da: e8 1c fc ff ff        callq efb <vuln_func>
12df: 48 8d 85 c0 fb ff ff  lea -0x440(%rbp),%rax
12e6: 48 89 c7              mov  %rax,%rdi
..... .....         .....
```

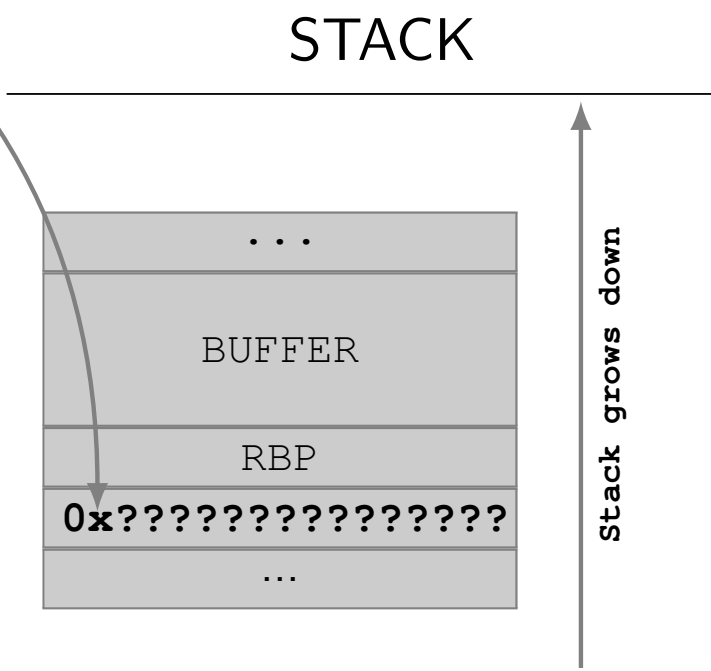


1) Extracting static information

→ Our goal is to obtain an address belonging to the application.
→ We are going to obtain the **saved-IP** of vulnerable function caller.

Offset2lib with **saved-IP** ⇒ **all** mmapped areas.

```
0000000000001063 <attend_client>:
1063: 55                    push %rbp
1064: 48 89 e5              mov  %rsp,%rbp
1067: 48 81 ec 60 04 00 00  sub  $0x460,%rsp
106e: 64 48 8b 04 25 28 00  mov  %fs:0x28,%rax
1075: 00 00
..... .....         .....
12d7: 48 89 c7              mov  %rax,%rdi
12da: e8 1c fc ff ff        callq efb <vuln_func>
12df: 48 8d 85 c0 fb ff ff  lea -0x440(%rbp),%rax
12e6: 48 89 c7              mov  %rax,%rdi
..... .....         .....
```

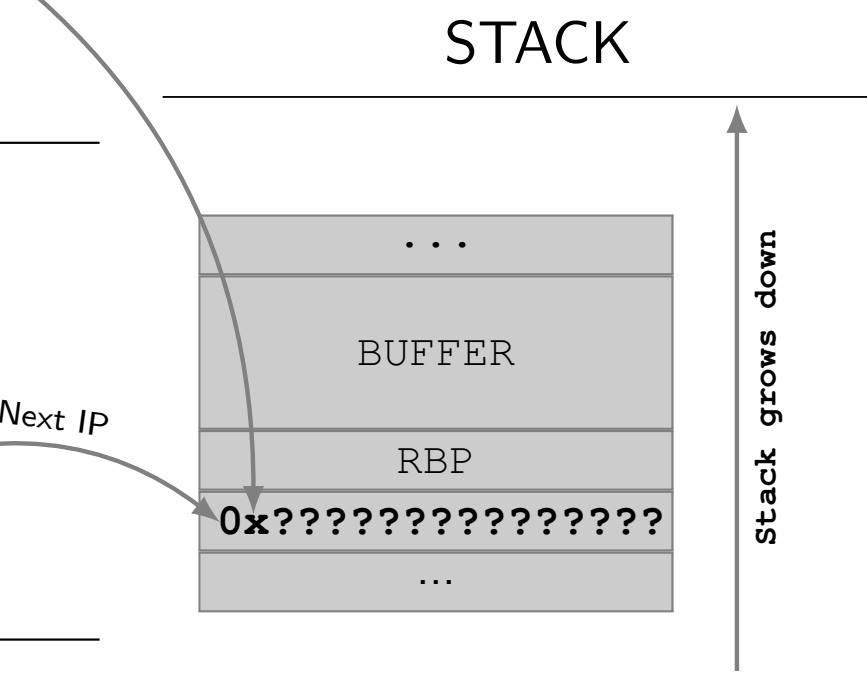


1) Extracting static information

→ Our goal is to obtain an address belonging to the application.
→ We are going to obtain the **saved-IP** of vulnerable function caller.

Offset2lib with **saved-IP** ⇒ **all** mmapped areas.

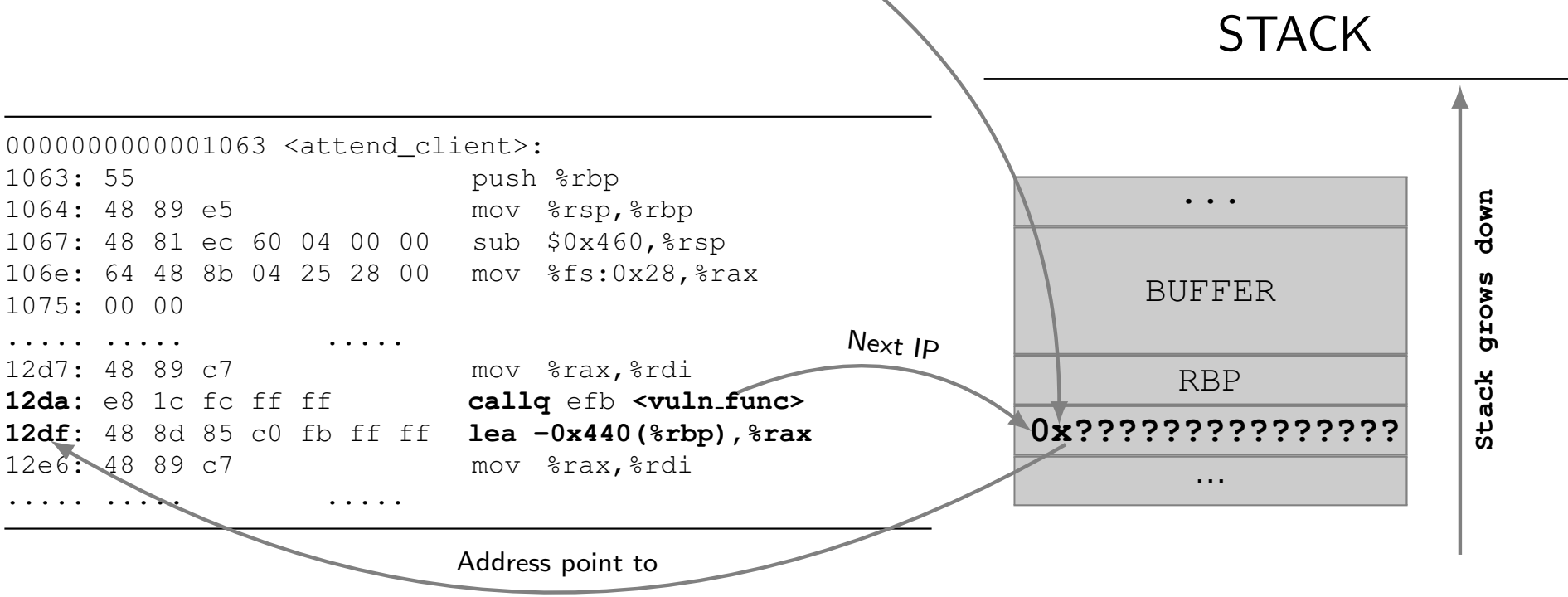
```
0000000000001063 <attend_client>:
1063: 55                    push %rbp
1064: 48 89 e5              mov  %rsp,%rbp
1067: 48 81 ec 60 04 00 00  sub  $0x460,%rsp
106e: 64 48 8b 04 25 28 00  mov  %fs:0x28,%rax
1075: 00 00
..... .....         .....
12d7: 48 89 c7              mov  %rax,%rdi
12da: e8 1c fc ff ff        callq efb <vuln_func>
12df: 48 8d 85 c0 fb ff ff  lea -0x440(%rbp),%rax
12e6: 48 89 c7              mov  %rax,%rdi
..... .....         .....
```



1) Extracting static information

→ Our goal is to obtain an address belonging to the application.
→ We are going to obtain the **saved-IP** of vulnerable function caller.

Offset2lib with **saved-IP** ⇒ **all** mmapped areas.

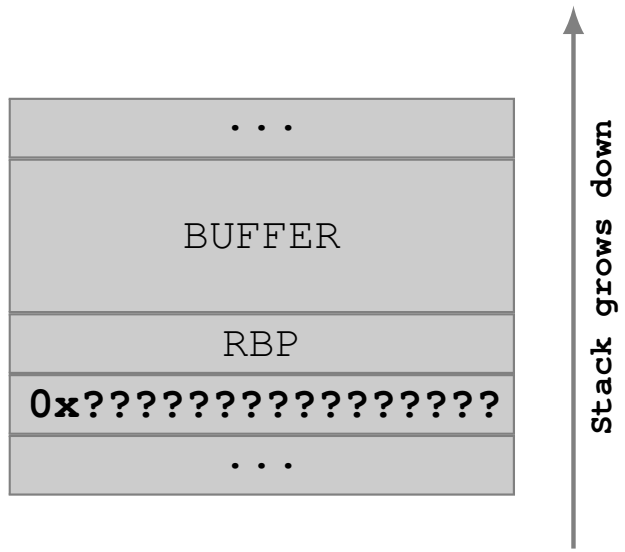


1) Extracting static information

Memory map

```
7fd1b414f000-7fd1b430a000 r-xp /lib/.../libc-2.19.so
7fd1b430a000-7fd1b450a000 ---p /lib/.../libc-2.19.so
7fd1b450a000-7fd1b450e000 r--p /lib/.../libc-2.19.so
7fd1b450e000-7fd1b4510000 rw-p /lib/.../libc-2.19.so
7fd1b4510000-7fd1b4515000 rw-p
7fd1b4515000-7fd1b4538000 r-xp /lib/.../ld-2.19.so
7fd1b4718000-7fd1b471b000 rw-p
7fd1b4734000-7fd1b4737000 rw-p
7fd1b4737000-7fd1b4738000 r--p /lib/.../ld-2.19.so
7fd1b4738000-7fd1b4739000 rw-p /lib/.../ld-2.19.so
7fd1b4739000-7fd1b473a000 rw-p
7fd1b473a000-7fd1b473c000 r-xp /root/server_64_PIE
7fd1b493b000-7fd1b493c000 r--p /root/server_64_PIE
7fd1b493c000-7fd1b493d000 rw-p /root/server_64_PIE
7fff981fa000-7fff9821b000 rw-p [stack]
7fff983fe000-7fff98400000 r-xp [vdso]
```

STACK



This value (`0x00007F`) can be obtained:

1. Running the application and showing the memory map.
2. Checking the source code if set any limit to stack.

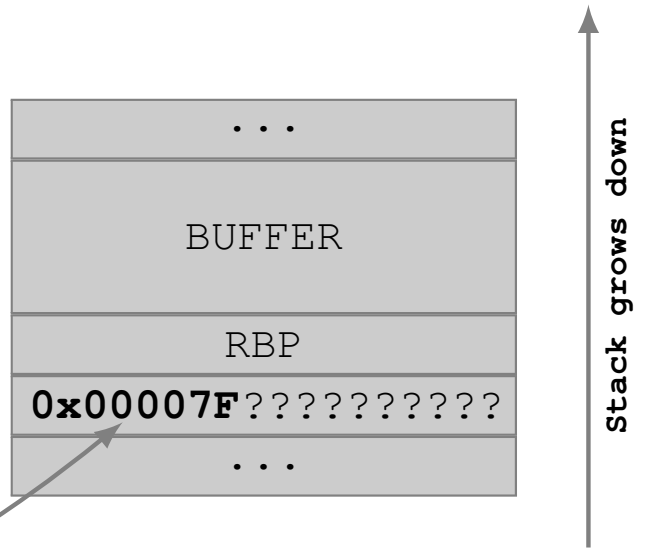
1) Extracting static information

Memory map

```
7fd1b414f000-7fd1b430a000 r-xp /lib/.../libc-2.19.so
7fd1b430a000-7fd1b450a000 ---p /lib/.../libc-2.19.so
7fd1b450a000-7fd1b450e000 r--p /lib/.../libc-2.19.so
7fd1b450e000-7fd1b4510000 rw-p /lib/.../libc-2.19.so
7fd1b4510000-7fd1b4515000 rw-p
7fd1b4515000-7fd1b4538000 r-xp /lib/.../ld-2.19.so
7fd1b4718000-7fd1b471b000 rw-p
7fd1b4734000-7fd1b4737000 rw-p
7fd1b4737000-7fd1b4738000 r--p /lib/.../ld-2.19.so
7fd1b4738000-7fd1b4739000 rw-p /lib/.../ld-2.19.so
7fd1b4739000-7fd1b473a000 rw-p
7fd1b473a000-7fd1b473c000 r-xp /root/server_64_PIE
7fd1b493b000-7fd1b493c000 r--p /root/server_64_PIE
7fd1b493c000-7fd1b493d000 rw-p /root/server_64_PIE
7fff981fa000-7fff9821b000 rw-p [stack]
7fff983fe000-7fff98400000 r-xp [vdso]
```

Highest 24 bits

STACK



This value (`0x00007F`) can be obtained:

1. Running the application and showing the memory map.
2. Checking the source code if set any limit to stack.

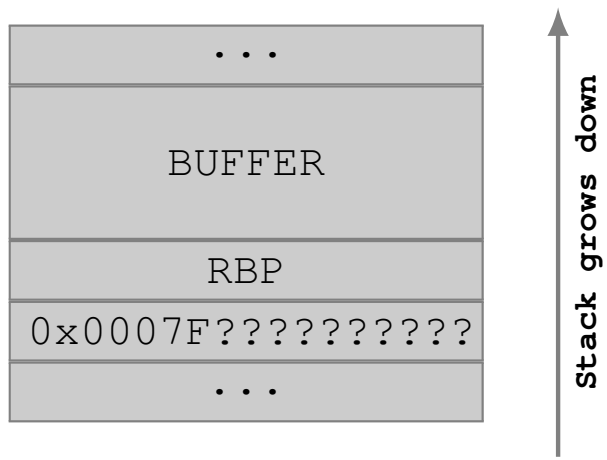
1) Extracting static information

Since the executable has to be `PAGE_SIZE` aligned, the 12 lower bits will not change when the executable is randomly loaded.

ASM Code

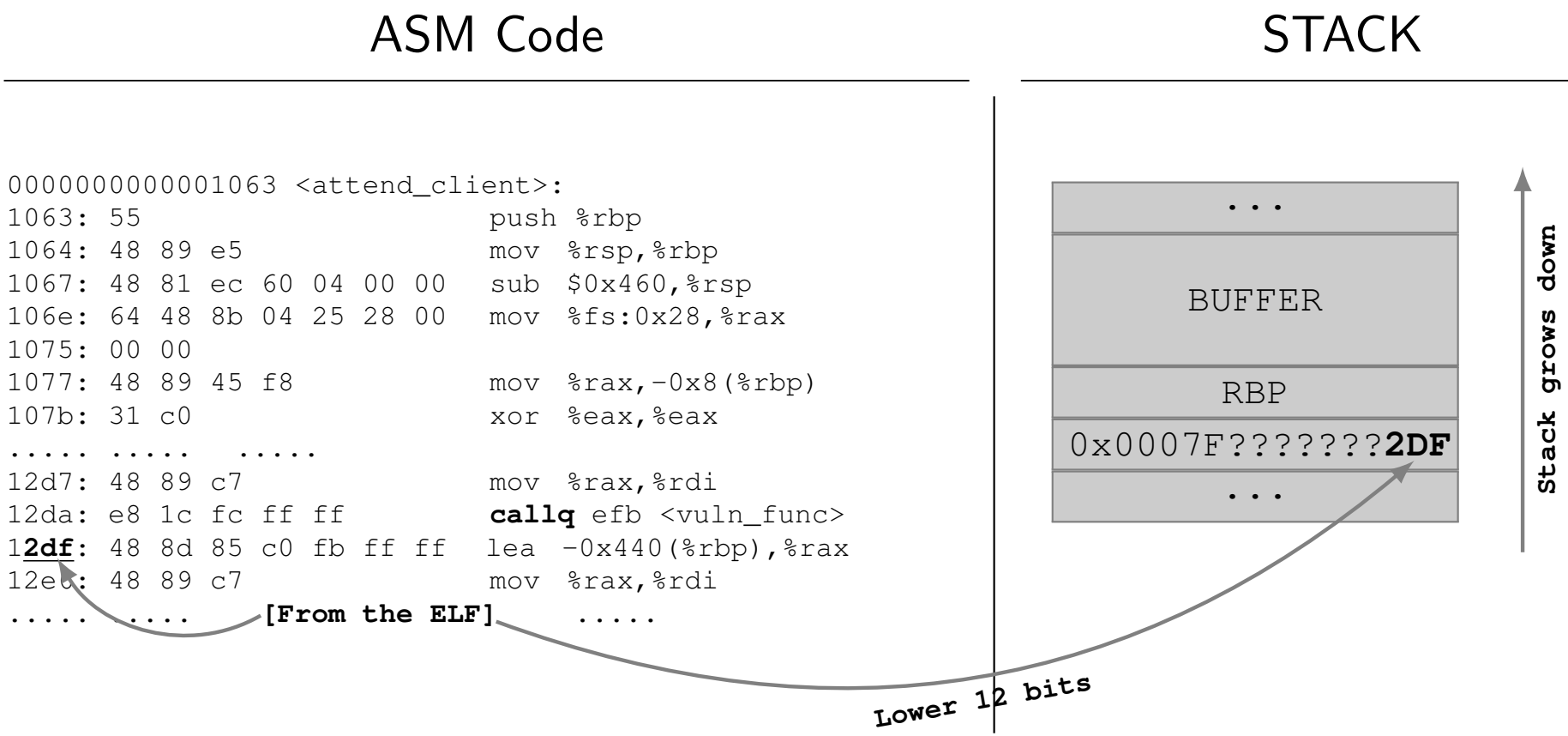
```
0000000000001063 <attend_client>:
1063: 55                    push %rbp
1064: 48 89 e5              mov  %rsp,%rbp
1067: 48 81 ec 60 04 00 00  sub  $0x460,%rsp
106e: 64 48 8b 04 25 28 00  mov  %fs:0x28,%rax
1075: 00 00
1077: 48 89 45 f8           mov  %rax,-0x8(%rbp)
107b: 31 c0                 xor  %eax,%eax
..... ..... .....
12d7: 48 89 c7              mov  %rax,%rdi
12da: e8 1c fc ff ff        callq efb <vuln_func>
12df: 48 8d 85 c0 fb ff ff  lea  -0x440(%rbp),%rax
12e6: 48 89 c7              mov  %rax,%rdi
..... .... [From the ELF] .....
```

STACK



1) Extracting static information

Since the executable has to be `PAGE_SIZE` aligned, the 12 lower bits will not change when the executable is randomly loaded.

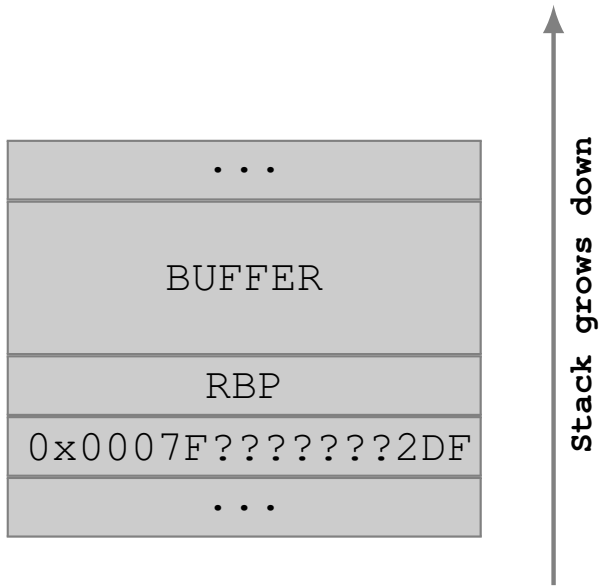


2) Brute forcing Saved-IP address

```
void vuln_func(char *str, int lstr){
  char buff[48];
  int i = 0;
  ...
  for (i = 0; i < lstr; i++) {
    if (str[i] != '\n')
      buff[lbuff++] = str[i];
  ...
}
```

- The unknown 28 random bits: "byte-for-byte" attack.
- The first byte is "special", we know the lowest 4 bits:
  - $0x?2_{16} \rightarrow ??10_2 \rightarrow 2^4 = 16$ attempts
  - {0x02, 0x12, 0x22 ... 0xC2, 0xD2, 0xE2, 0xF2}

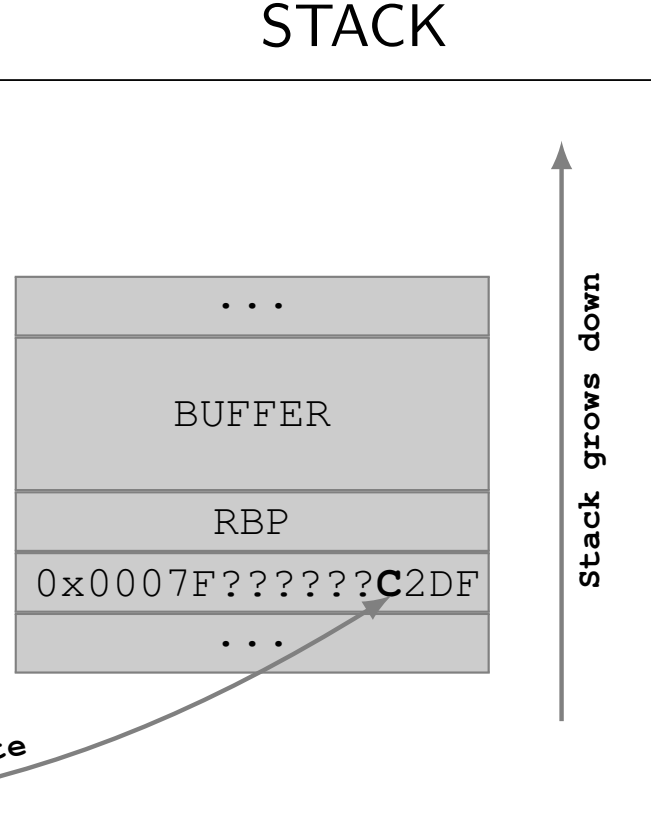
STACK



2) Brute forcing Saved-IP address

```
void vuln_func(char *str, int lstr){
  char buff[48];
  int i = 0;
  ...
  for (i = 0; i < lstr; i++) {
    if (str[i] != '\n')
      buff[lbuff++] = str[i];
  ...
}
```

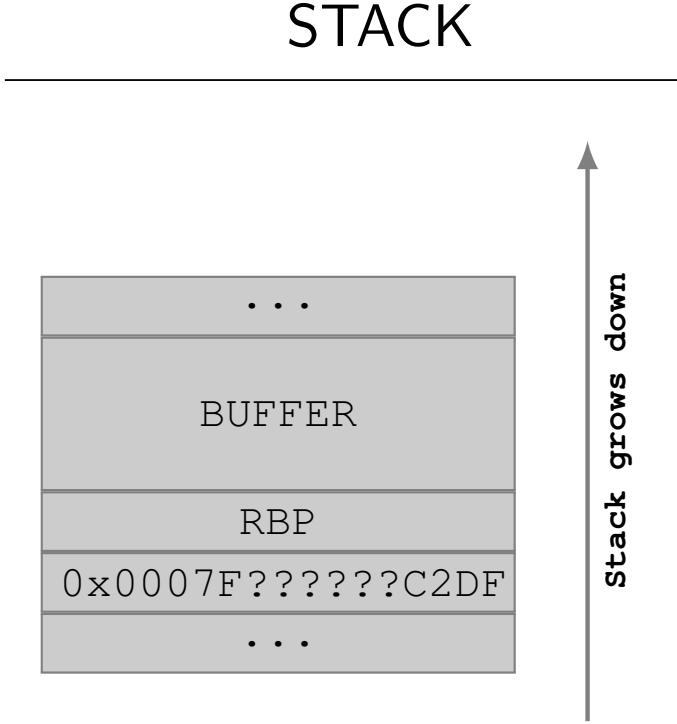
- The unknown 28 random bits: "byte-for-byte" attack.
- The first byte is "special", we know the lowest 4 bits:
  - $0x?2_{16} \rightarrow ??10_2 \rightarrow 2^4 = 16$ attempts
  - {0x02, 0x12, 0x22 ... **0xC2**, 0xD2, 0xE2, 0xF2}



2) Brute forcing Saved-IP address

```
void vuln_func(char *str, int lstr){
  char buff[48];
  int i = 0;
  ...
  for (i = 0; i < lstr; i++) {
    if (str[i] != '\n')
      buff[lbuff++] = str[i];
  ...
}
```

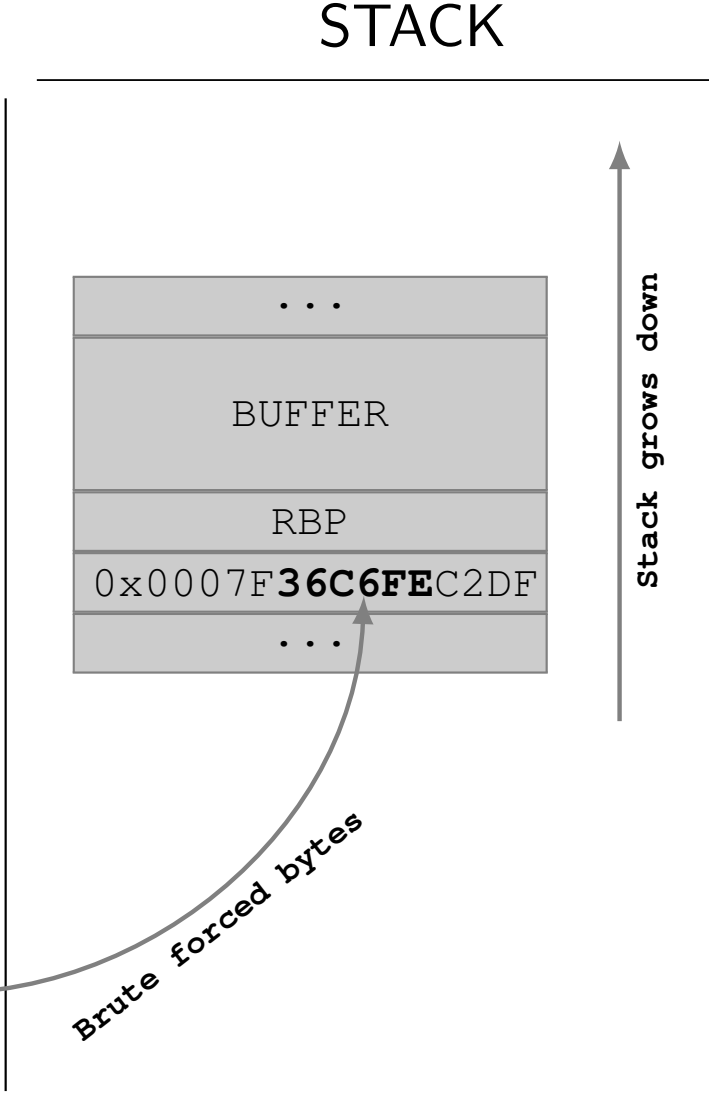
- The unknown 28 random bits: "byte-for-byte" attack.
- The first byte is "special", we know the lowest 4 bits:
  - $0x?2_{16} \rightarrow ??10_2 \rightarrow 2^4 = 16$ attempts
  - {0x02, 0x12, 0x22 ... **0xC2**, 0xD2, 0xE2, 0xF2}
- The remaining 3 bytes $\rightarrow$ standard "byte-for-byte" attack
  - $3x2^8 = 768$ attempts.
- After execute the byte-for-byte we obtained 0x36C6FE



2) Brute forcing Saved-IP address

```
void vuln_func(char *str, int lstr){
  char buff[48];
  int i = 0;
  ...
  for (i = 0; i < lstr; i++) {
    if (str[i] != '\n')
      buff[lbuff++] = str[i];
  ...
}
```

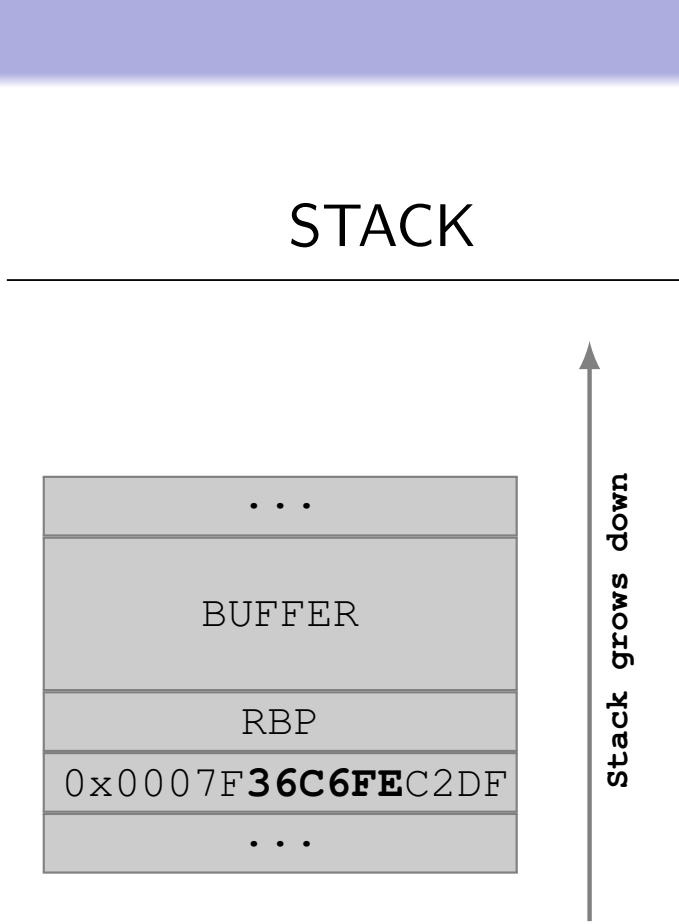
- The unknown 28 random bits: "byte-for-byte" attack.
- The first byte is "special", we know the lowest 4 bits:
  - $0x?2_{16} \rightarrow ??10_2 \rightarrow 2^4 = 16$ attempts
  - {0x02, 0x12, 0x22 ... **0xC2**, 0xD2, 0xE2, 0xF2}
- The remaining 3 bytes $\rightarrow$ standard "byte-for-byte" attack
  - $3x2^8 = 768$ attempts.
- After execute the byte-for-byte we obtained **0x36C6FE**



2) Brute forcing Saved-IP address

```
void vuln_func(char *str, int lstr){
  char buff[48];
  int i = 0;
  ...
  for (i = 0; i < lstr; i++) {
    if (str[i] != '\n')
      buff[lbuff++] = str[i];
  ...
}
```

- The unknown 28 random bits: "byte-for-byte" attack.
- The first byte is "special", we know the lowest 4 bits:
  - $0x?2_{16} \rightarrow ??10_2 \rightarrow 2^4 = 16$ attempts
  - {0x02, 0x12, 0x22 ... **0xC2**, 0xD2, 0xE2, 0xF2}
- The remaining 3 bytes $\rightarrow$ standard "byte-for-byte" attack
  - $3x2^8 = 768$ attempts.
- After execute the byte-for-byte we obtained **0x36C6FE**
- We need to perform $\frac{2^4+3*2^8}{2} = 392$ attempts on average.

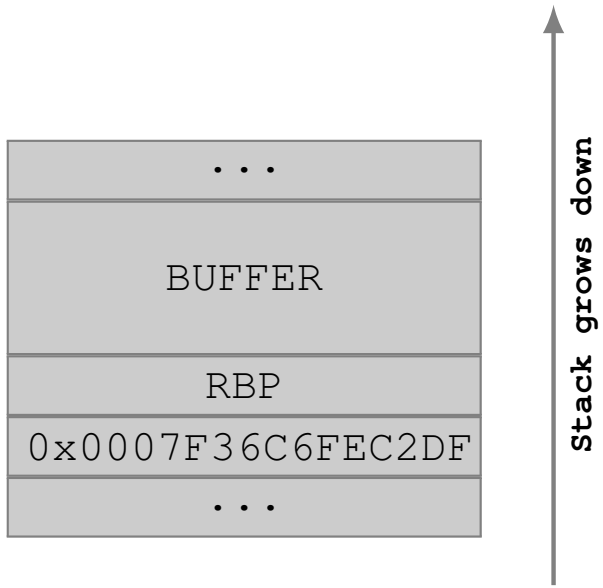


3) Calculating base application address

```
0000000000001063 <attend_client>:
1063: 55                    push %rbp
1064: 48 89 e5              mov  %rsp,%rbp
1067: 48 81 ec 60 04 00 00  sub  $0x460,%rsp
106e: 64 48 8b 04 25 28 00  mov  %fs:0x28,%rax
1075: 00 00
1077: 48 89 45 f8           mov  %rax,-0x8(%rbp)
107b: 31 c0                 xor  %eax,%eax
..... .....   .....
12d7: 48 89 c7              mov  %rax,%rdi
12da: e8 1c fc ff ff        callq efb <vuln_func>
12df: 48 8d 85 c0 fb ff ff  lea  -0x440(%rbp),%rax
12e6: 48 89 c7              mov  %rax,%rdi
..... .....         .....
```

App_base=(savedIP & 0xFFF)-(CALLER_PAGE_OFFSET << 12)

STACK

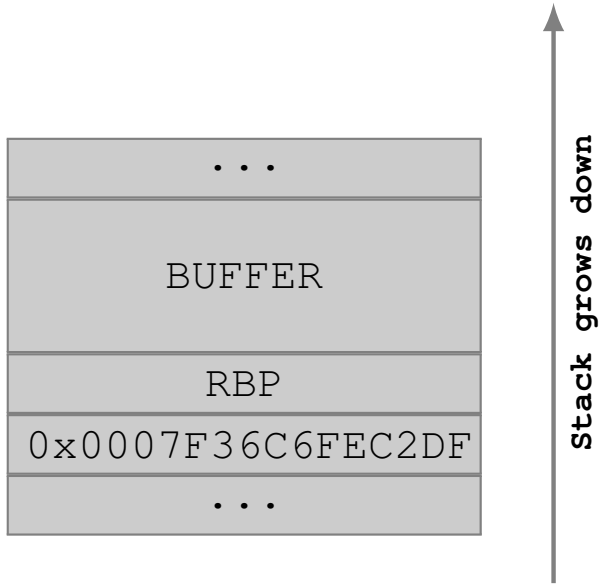


3) Calculating base application address

```
0000000000001063 <attend_client>:
1063: 55                    push %rbp
1064: 48 89 e5              mov  %rsp,%rbp
1067: 48 81 ec 60 04 00 00  sub  $0x460,%rsp
106e: 64 48 8b 04 25 28 00  mov  %fs:0x28,%rax
1075: 00 00
1077: 48 89 45 f8           mov  %rax,-0x8(%rbp)
107b: 31 c0                 xor  %eax,%eax
..... .....   .....
12d7: 48 89 c7              mov  %rax,%rdi
12da: e8 1c fc ff ff        callq efb <vuln_func>
12df: 48 8d 85 c0 fb ff ff  lea  -0x440(%rbp),%rax
12e6: 48 89 c7              mov  %rax,%rdi
..... .....         .....
```

App_base=(savedIP & 0xFFF)-(CALLER_PAGE_OFFSET << 12)

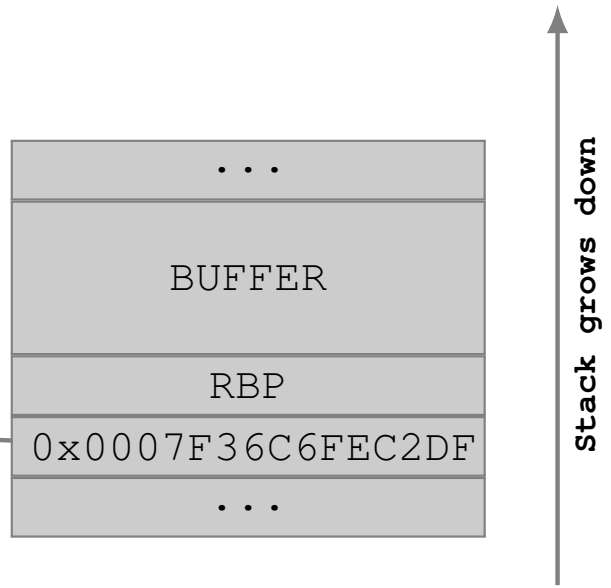
STACK



3) Calculating base application address

```
0000000000001063 <attend_client>:
1063: 55                    push %rbp
1064: 48 89 e5              mov  %rsp,%rbp
1067: 48 81 ec 60 04 00 00  sub  $0x460,%rsp
106e: 64 48 8b 04 25 28 00  mov  %fs:0x28,%rax
1075: 00 00
1077: 48 89 45 f8           mov  %rax,-0x8(%rbp)
107b: 31 c0                 xor  %eax,%eax
..... .....   .....
12d7: 48 89 c7              mov  %rax,%rdi
12da: e8 1c fc ff ff        callq efb <vuln_func>
12df: 48 8d 85 c0 fb ff ff  lea  -0x440(%rbp),%rax
12e6: 48 89 c7              mov  %rax,%rdi
..... .....         .....
```

STACK

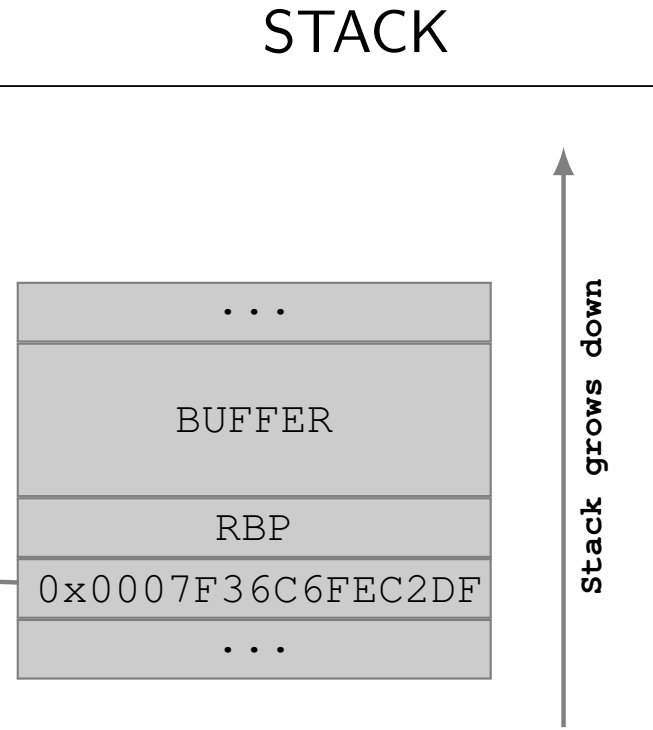


App_base=(savedIP & 0xFFF)-(CALLER_PAGE_OFFSET << 12)

0x7F36C6fEB000=(0x7f36C6FEC2DF & ~0xFFF)-(0x1000)

3) Calculating base application address

```
0000000000001063 <attend_client>:
1063: 55                    push %rbp
1064: 48 89 e5              mov  %rsp,%rbp
1067: 48 81 ec 60 04 00 00  sub  $0x460,%rsp
106e: 64 48 8b 04 25 28 00  mov  %fs:0x28,%rax
1075: 00 00
1077: 48 89 45 f8           mov  %rax,-0x8(%rbp)
107b: 31 c0                 xor  %eax,%eax
..... .....   .....
12d7: 48 89 c7              mov  %rax,%rdi
12da: e8 1c fc ff ff        callq efb <vuln_func>
12df: 48 8d 85 c0 fb ff ff  lea  -0x440(%rbp),%rax
12e6: 48 89 c7              mov  %rax,%rdi
..... .....         .....
```

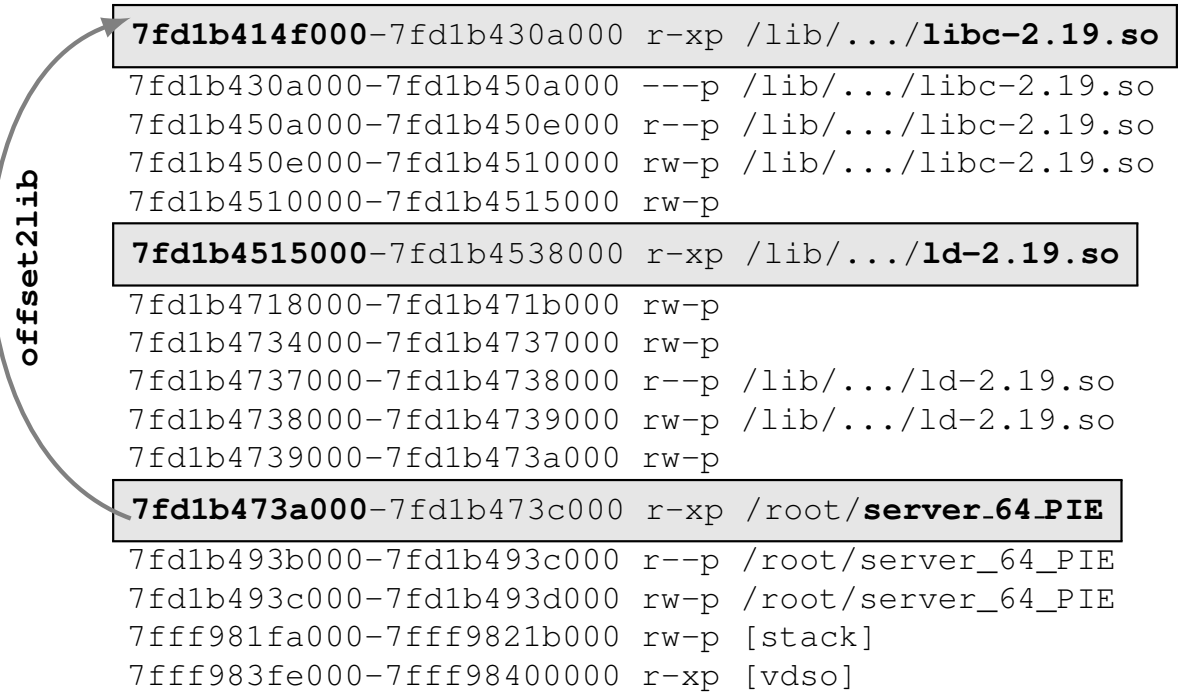


App_base=(savedIP & 0xFFF)-(CALLER_PAGE_OFFSET << 12)

0x7F36C6fEB000=(0x7f36C6FEC2DF & ~0xFFF)-(0x1000)

App. Base = 0x7F36C6fEB000

4) Calculating library offsets

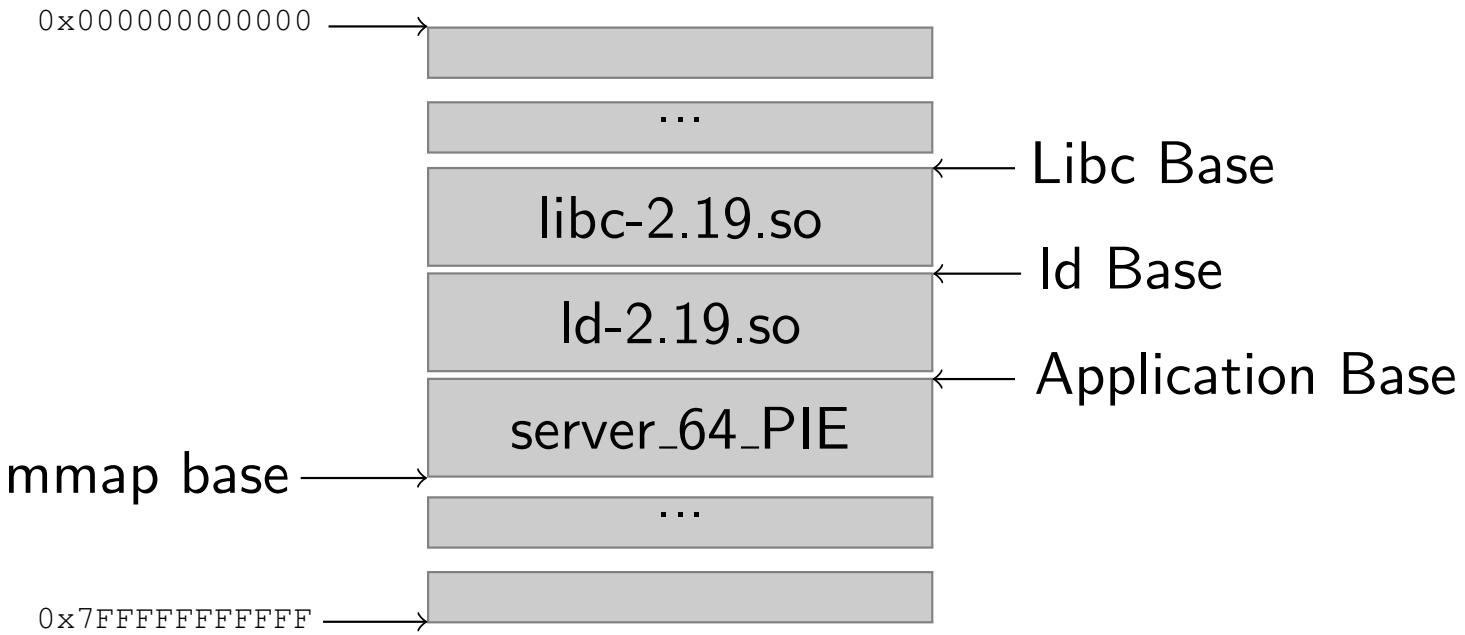


Distribution	Libc version	Offset2lib (bytes)
CentOS 6.5	2.12	0x5b6000
Debian 7.1	2.13	0x5ac000
Ubuntu 12.04 LTS	2.15	0x5e4000
Ubuntu 12.10	2.15	0x5e4000
Ubuntu 13.10	2.17	0x5ed000
openSUSE 13.1	2.18	0x5d1000
Ubuntu 14.04.1 LTS	2.19	0x5eb000

5) Getting app. process mapping

Obtaining library base addresses:

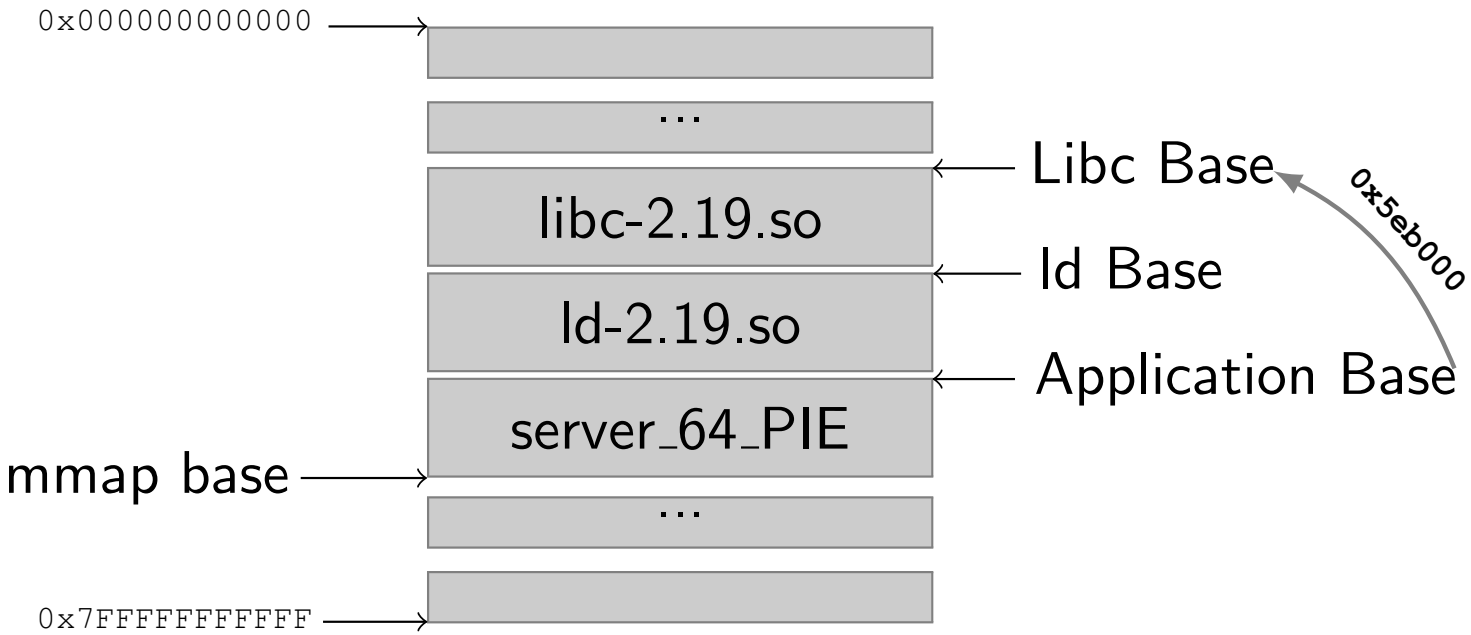
- Application Base = `0x7FD1B473A000`
- Offset2lib (libc) = `0x5eb000`
- Offset2lib (ld) = `0x225000`



5) Getting app. process mapping

Obtaining library base addresses:

- Application Base = `0x7FD1B473A000`
- Offset2lib (libc) = `0x5eb000`
- Offset2lib (ld) = `0x225000`

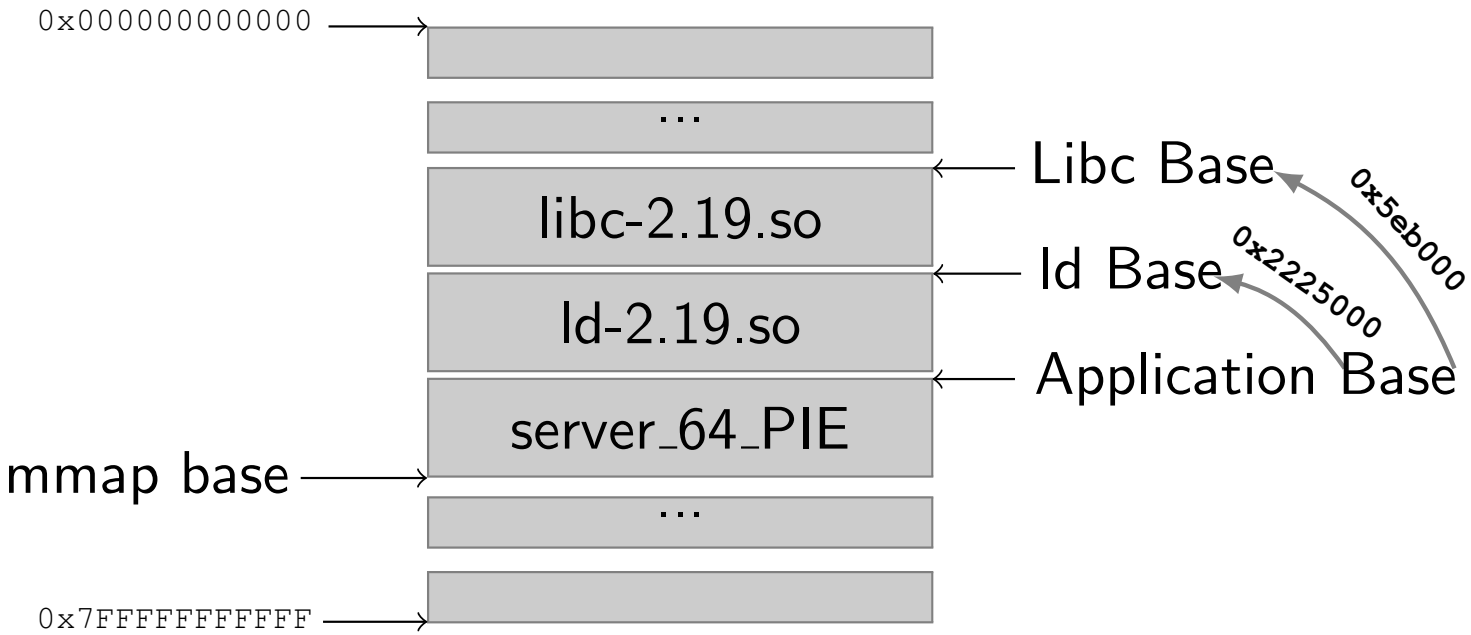


Libc Base = `0x7FD1B473A000` - `0x5eb000` = **`0x7FD1B414F000`**

5) Getting app. process mapping

Obtaining library base addresses:

- Application Base = `0x7FD1B473A000`
- Offset2lib (libc) = `0x5eb000`
- Offset2lib (ld) = `0x225000`



Libc Base = `0x7FD1B473A000 - 0x5eb000` = **`0x7FD1B414F000`**

ld Base = `0x7FD1B473A000 - 0x225000` = **`0x7fd1b4515000`**

The vulnerable server

To show a more realistic PoC:

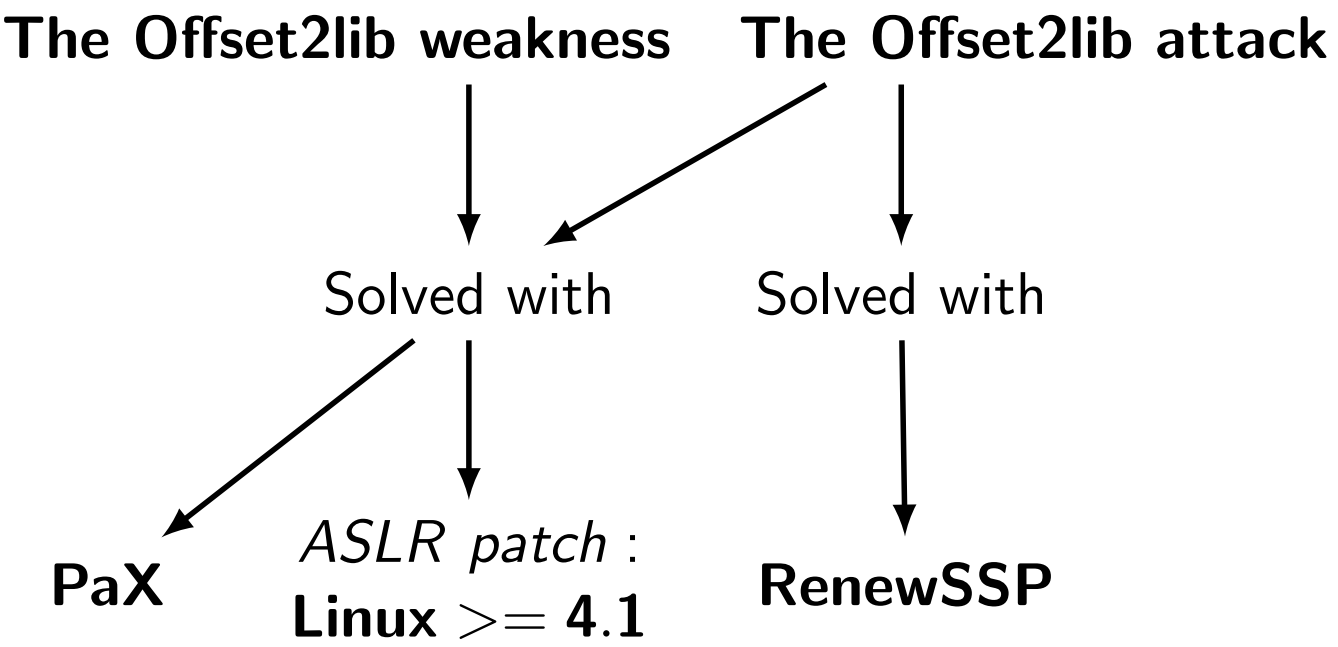
- Bypass NX, SSP, ASLR, FORTIFY or RELRO.
- We do not use GOT neither PLT.
- Valid for any application (Gadgets only from libraries)
- We use a fully updated Linux, Ubuntu 14.04.4 LTS (trusty).

Parameter	Comment	Configuration
App. relocatable	Yes	`-fpie -pie`
Lib. relocatable	Yes	`-Fpic`
ASLR config.	Enabled	`randomize_va_space = 2`
SSP	Enabled	`-fstack-protector-all`
Arch.	64 bits	x86_64 GNU/Linux
NX	Enabled	PAE or x64
RELRO	Full	`-Wl,-z,-relro,-z,now`
FORTIFY	Yes	`-D_FORTIFY_SOURCE=2`
Optimisation	Yes	`-O2`

Bypassing NX, SSP and ASLR on 64-bit Linux

Demo: Bypass NX, SSP and ASLR in < 1 sec.

Solutions overview



All weaknesses are only solved by the ASLR-NG

With Linux Kernel < 4.1

```
# echo 2 > /proc/sys/kernel/randomize_va_space
# hello_world_dynamic_pie
7f621ffbb000-7f6220176000 r-xp 00000000 00:02 5192 /lib/x86_64-linux-gnu/libc.so.6
7f6220176000-7f6220376000 ---p 001bb000 00:02 5192 /lib/x86_64-linux-gnu/libc.so.6
7f6220376000-7f622037a000 r--p 001bb000 00:02 5192 /lib/x86_64-linux-gnu/libc.so.6
7f622037a000-7f622037c000 rw-p 001bf000 00:02 5192 /lib/x86_64-linux-gnu/libc.so.6
7f622037c000-7f6220381000 rw-p 00000000 00:00 0
7f6220381000-7f62203a4000 r-xp 00000000 00:02 4917 /lib64/ld-linux-x86-64.so.2
7f622059c000-7f622059d000 rw-p 00000000 00:00 0
7f622059d000-7f622059e000 r-xp 00000000 00:00 0
7f622059e000-7f62205a3000 rw-p 00000000 00:00 0
7f62205a3000-7f62205a4000 r--p 00022000 00:02 4917 /lib64/ld-linux-x86-64.so.2
7f62205a4000-7f62205a5000 rw-p 00023000 00:02 4917 /lib64/ld-linux-x86-64.so.2
7f62205a5000-7f62205a6000 rw-p 00000000 00:00 0
7f62205a6000-7f62205a7000 r-xp 00000000 00:02 4896 /bin/hello_world_dynamic_pie
7f62207a6000-7f62207a7000 r--p 00000000 00:02 4896 /bin/hello_world_dynamic_pie
7f62207a7000-7f62207a8000 rw-p 00001000 00:02 4896 /bin/hello_world_dynamic_pie
7fff47e15000-7fff47e36000 rw-p 00000000 00:00 0 [stack]
7fff47e63000-7fff47e65000 r--p 00000000 00:00 0 [vvar]
7fff47e65000-7fff47e67000 r-xp 00000000 00:00 0 [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

With Linux Kernel >= 4.1

```
# hello_world_dynamic_pie
54859ccd6000-54859ccd7000 r-xp 00000000 00:02 4896 /bin/hello_world_dynamic_pie
54859ced6000-54859ced7000 r--p 00000000 00:02 4896 /bin/hello_world_dynamic_pie
54859ced7000-54859ced8000 rw-p 00001000 00:02 4896 /bin/hello_world_dynamic_pie
7f75be764000-7f75be91f000 r-xp 00000000 00:02 5192 /lib/x86_64-linux-gnu/libc.so.6
7f75be91f000-7f75beb1f000 ---p 001bb000 00:02 5192 /lib/x86_64-linux-gnu/libc.so.6
7f75beb1f000-7f75beb23000 r--p 001bb000 00:02 5192 /lib/x86_64-linux-gnu/libc.so.6
7f75beb23000-7f75beb25000 rw-p 001bf000 00:02 5192 /lib/x86_64-linux-gnu/libc.so.6
7f75beb25000-7f75beb2a000 rw-p 00000000 00:00 0
7f75beb2a000-7f75beb4d000 r-xp 00000000 00:02 4917 /lib64/ld-linux-x86-64.so.2
7f75bed45000-7f75bed46000 rw-p 00000000 00:00 0
7f75bed46000-7f75bed47000 r-xp 00000000 00:00 0
7f75bed47000-7f75bed4c000 rw-p 00000000 00:00 0
7f75bed4c000-7f75bed4d000 r--p 00022000 00:02 4917 /lib64/ld-linux-x86-64.so.2
7f75bed4d000-7f75bed4e000 rw-p 00023000 00:02 4917 /lib64/ld-linux-x86-64.so.2
7f75bed4e000-7f75bed4f000 rw-p 00000000 00:00 0
7fffb3741000-7fffb3762000 rw-p 00000000 00:00 0 [stack]
7fffb377b000-7fffb377d000 r--p 00000000 00:00 0 [vvar]
7fffb377d000-7fffb377f000 r-xp 00000000 00:00 0 [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Addressing the ASLR weaknesses

ASLR-NG addresses all these weaknesses but because of the urgency to fix the **Offset2lib** weakness, it was fixed in current Linux.

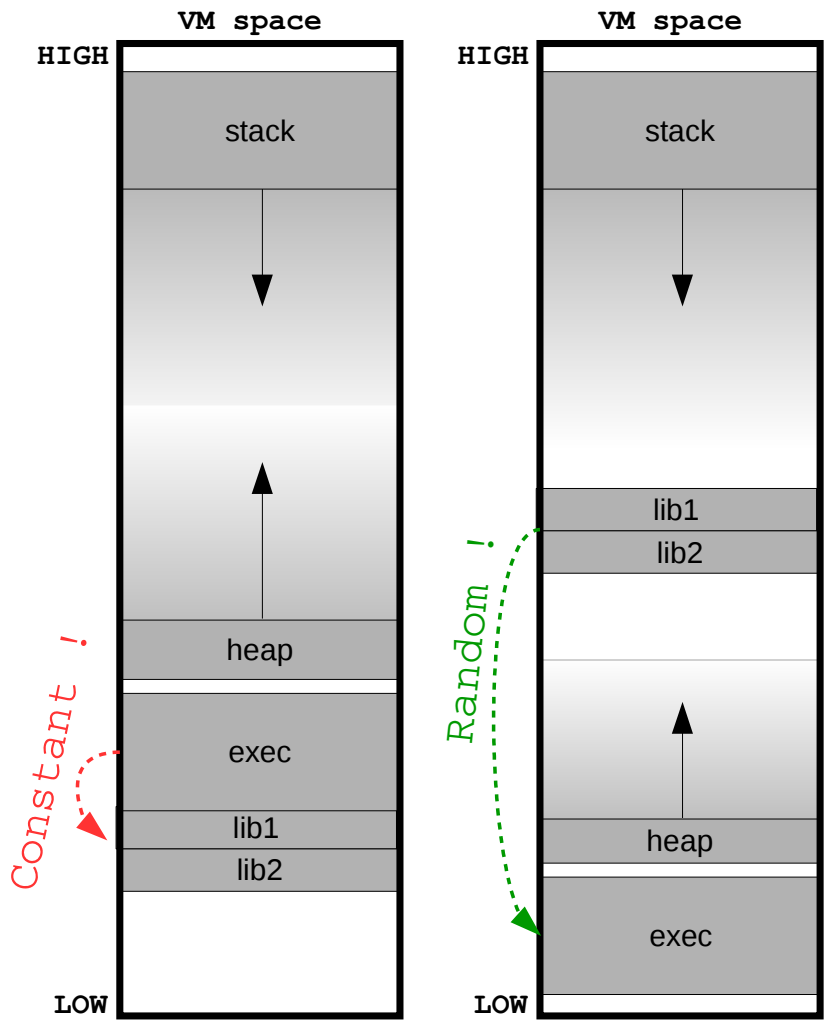
- It does not remove the correlation problem between all objects.

How we addressed the Offset2lib weakness ?

The particular Offset2lib fix

Offset2lib fix:

We have removed the correlation between the executable $\leftrightarrow$ libraries



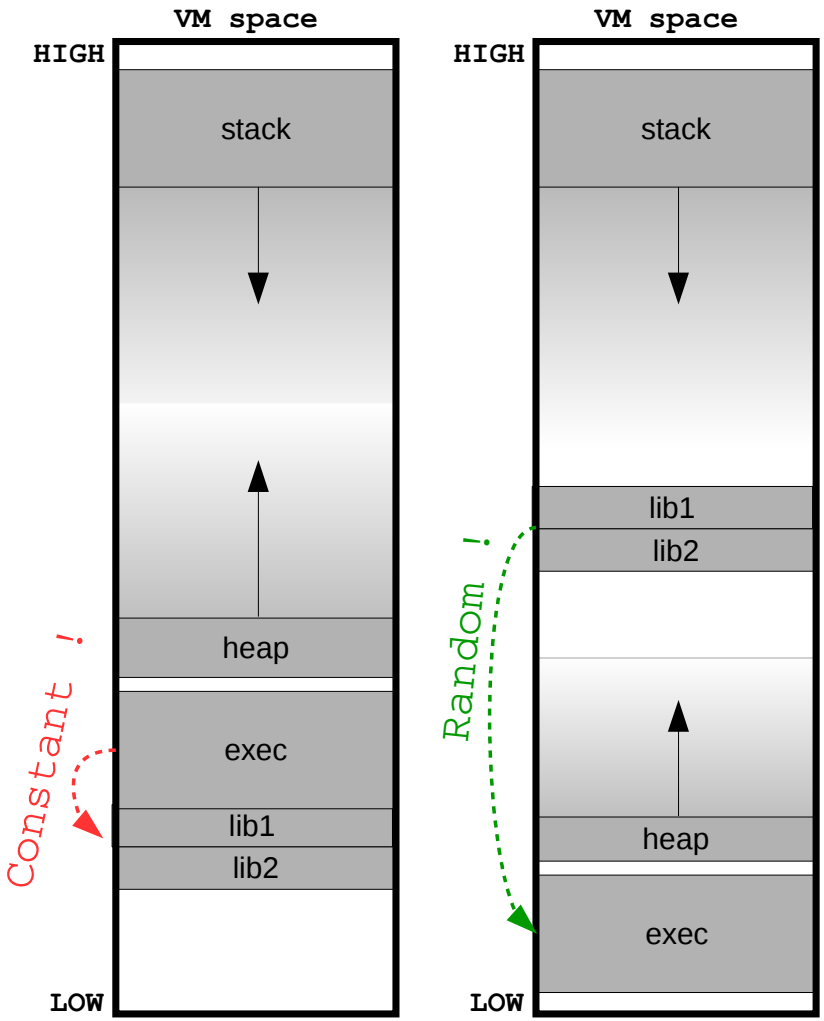
The particular Offset2lib fix

Offset2lib fix:

We have removed the correlation between the executable $\leftrightarrow$ libraries

Attack rewarded by Packet Storm Securit

Offset2lib was classified as 1-day vulnerability



The particular Offset2lib fix

Offset2lib fix:

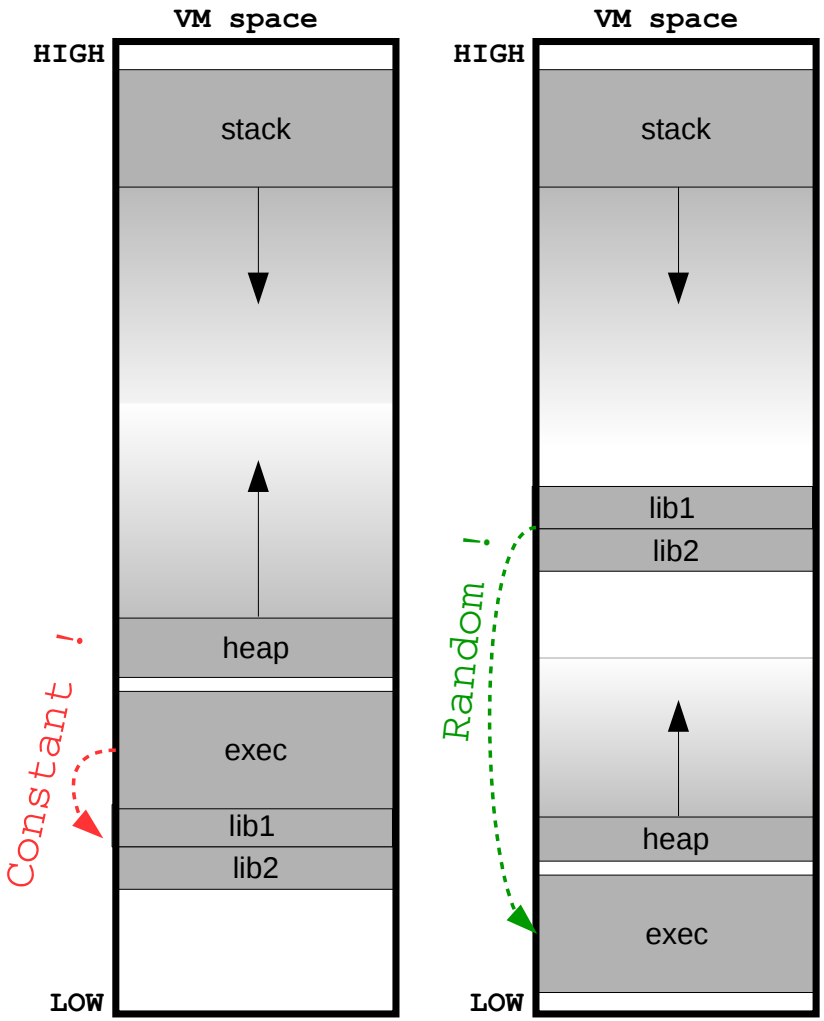
We have removed the correlation between the executable $\leftrightarrow$ libraries

Attack rewarded by Packet Storm Securit

Offset2lib was classified as 1-day vulnerability

Linux Kernel 4.1 patch:

We have created and sent a patch to Linux, which was considered **urgent**.

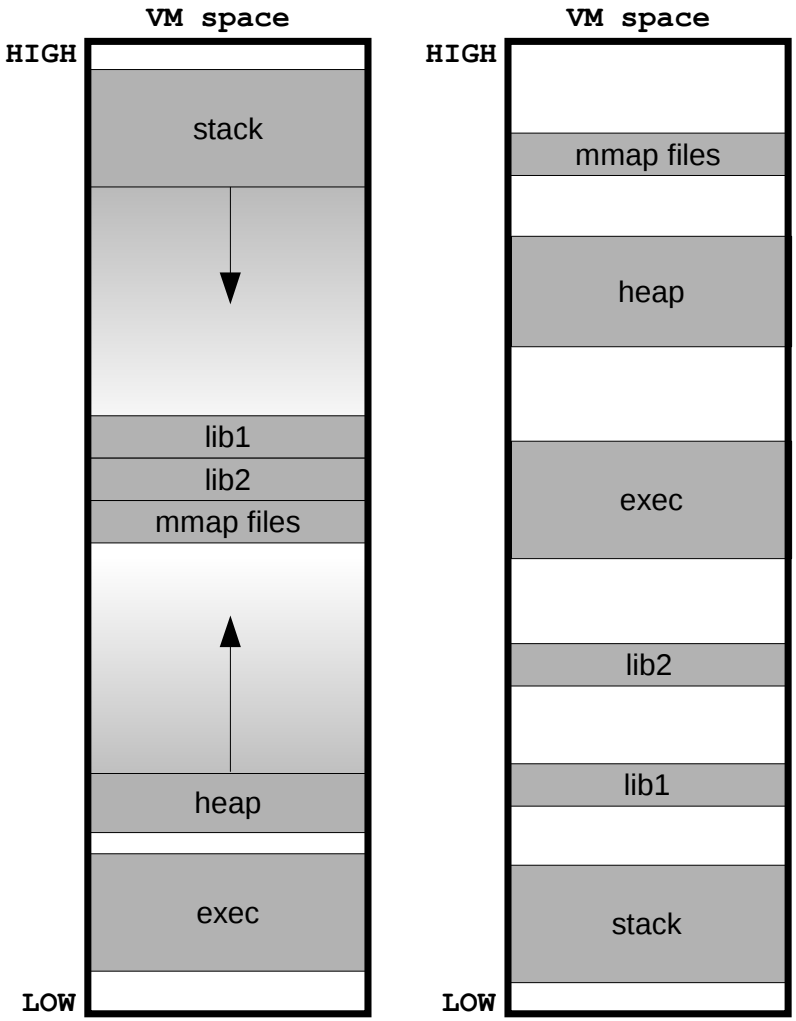


ASLR-NG: Address Space Layout Randomization Next Generation

ASLR-NG: The core ideas

A deep analysis of growable objects shows that they (stack and heap) can be bounded.

This key idea allows to load objects:



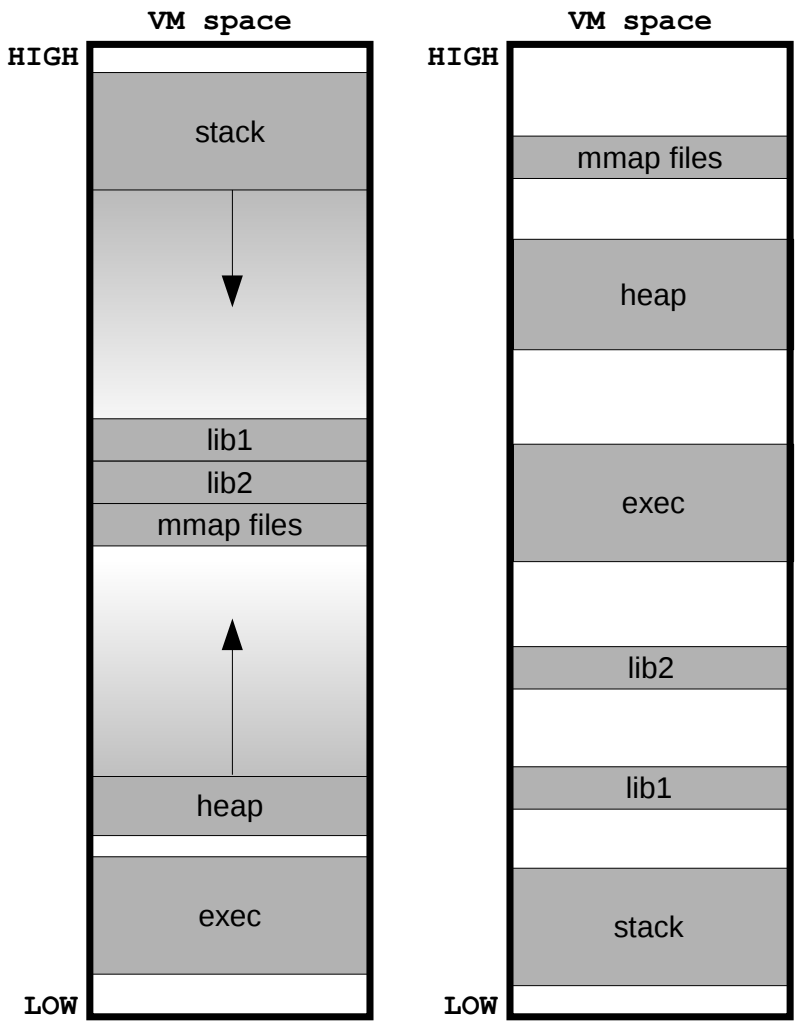
Linux ASLR Linux ASLR-NG

ASLR-NG: The core ideas

A deep analysis of growable objects shows that they (stack and heap) can be bounded.

This key idea allows to load objects:

- Freely along the VM:
  → Huge increment of entropy.



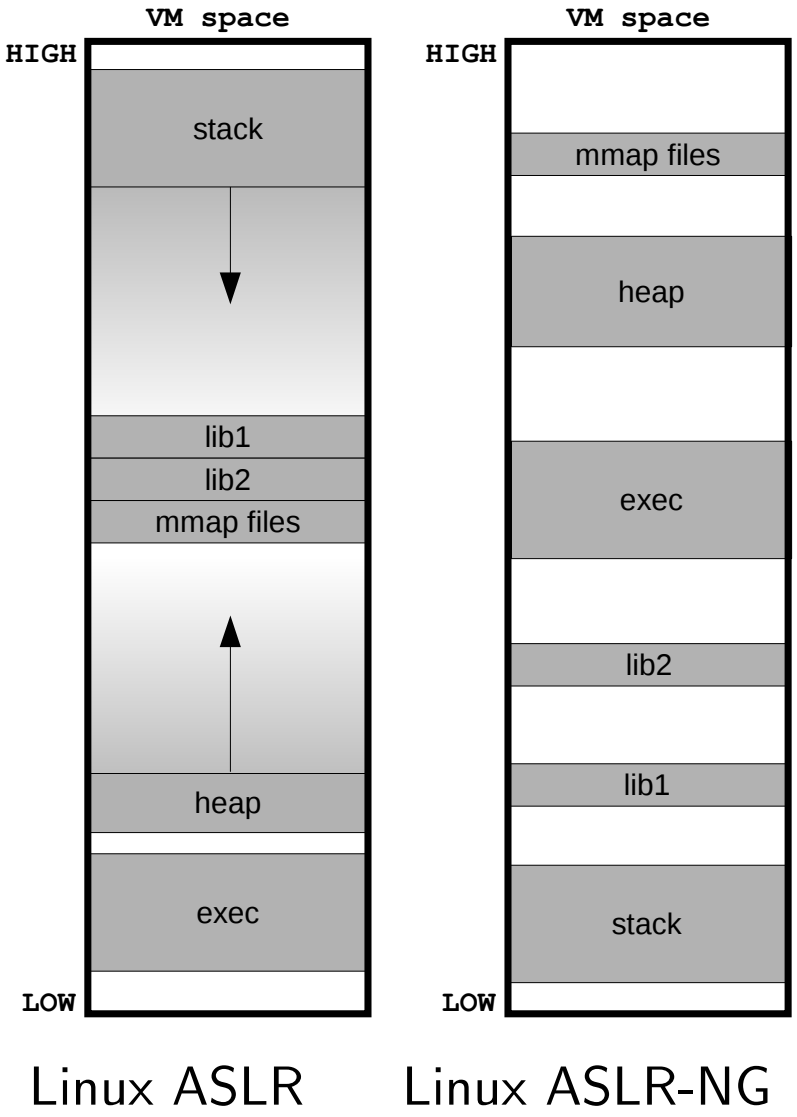
Linux ASLR Linux ASLR-NG

ASLR-NG: The core ideas

A deep analysis of growable objects shows that they (stack and heap) can be bounded.

This key idea allows to load objects:

- Freely along the VM:
  → Huge increment of entropy.
- Uniformly distributed:
  → No more likely addresses.



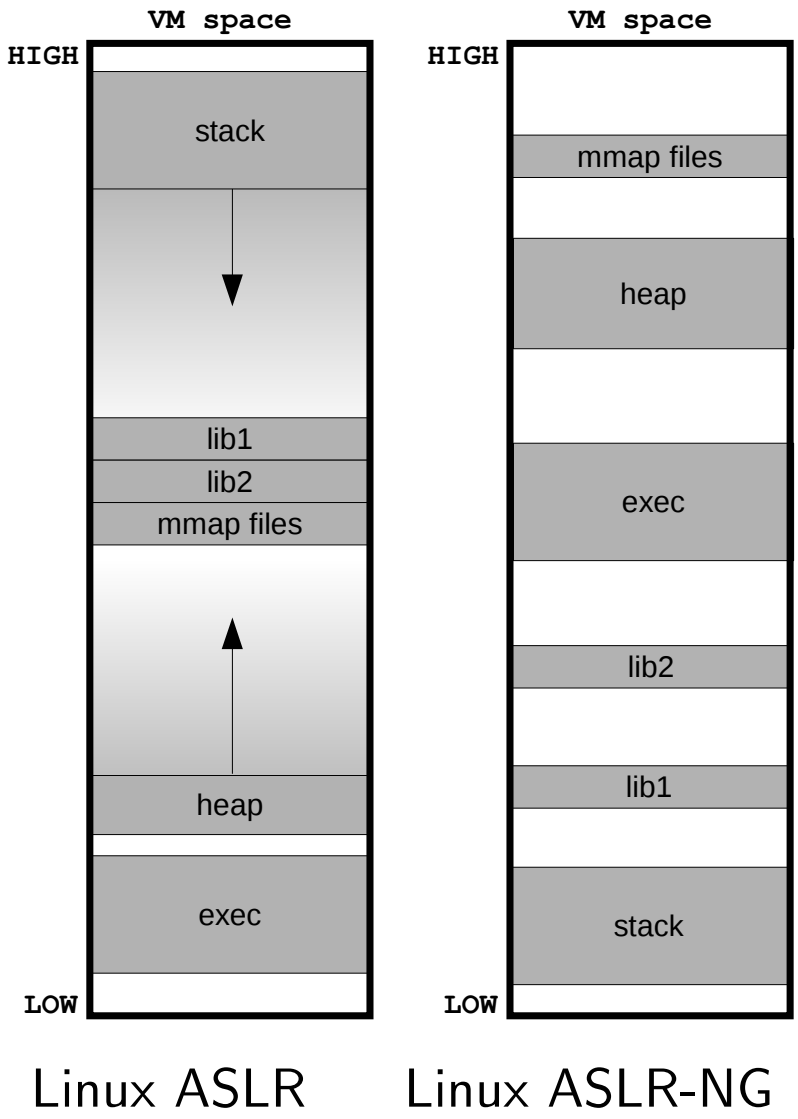
Linux ASLR Linux ASLR-NG

ASLR-NG: The core ideas

A deep analysis of growable objects shows that they (stack and heap) can be bounded.

This key idea allows to load objects:

- Freely along the VM:
  → Huge increment of entropy.
- Uniformly distributed:
  → No more likely addresses.
- Uncorrelated:
  → No more correlated attacks.



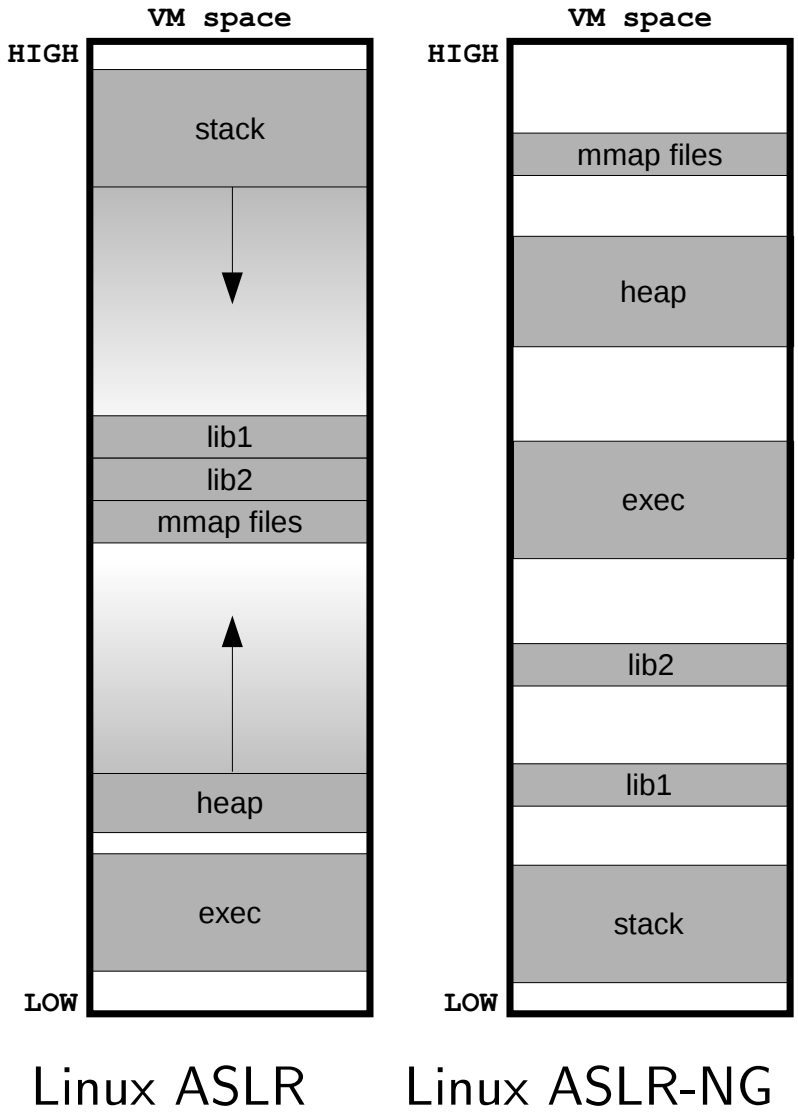
Linux ASLR Linux ASLR-NG

ASLR-NG: The core ideas

A deep analysis of growable objects shows that they (stack and heap) can be bounded.

This key idea allows to load objects:

- Freely along the VM:
  → Huge increment of entropy.
- Uniformly distributed:
  → No more likely addresses.
- Uncorrelated:
  → No more correlated attacks.
- Have different VM layout:
  → Forking model more secure.



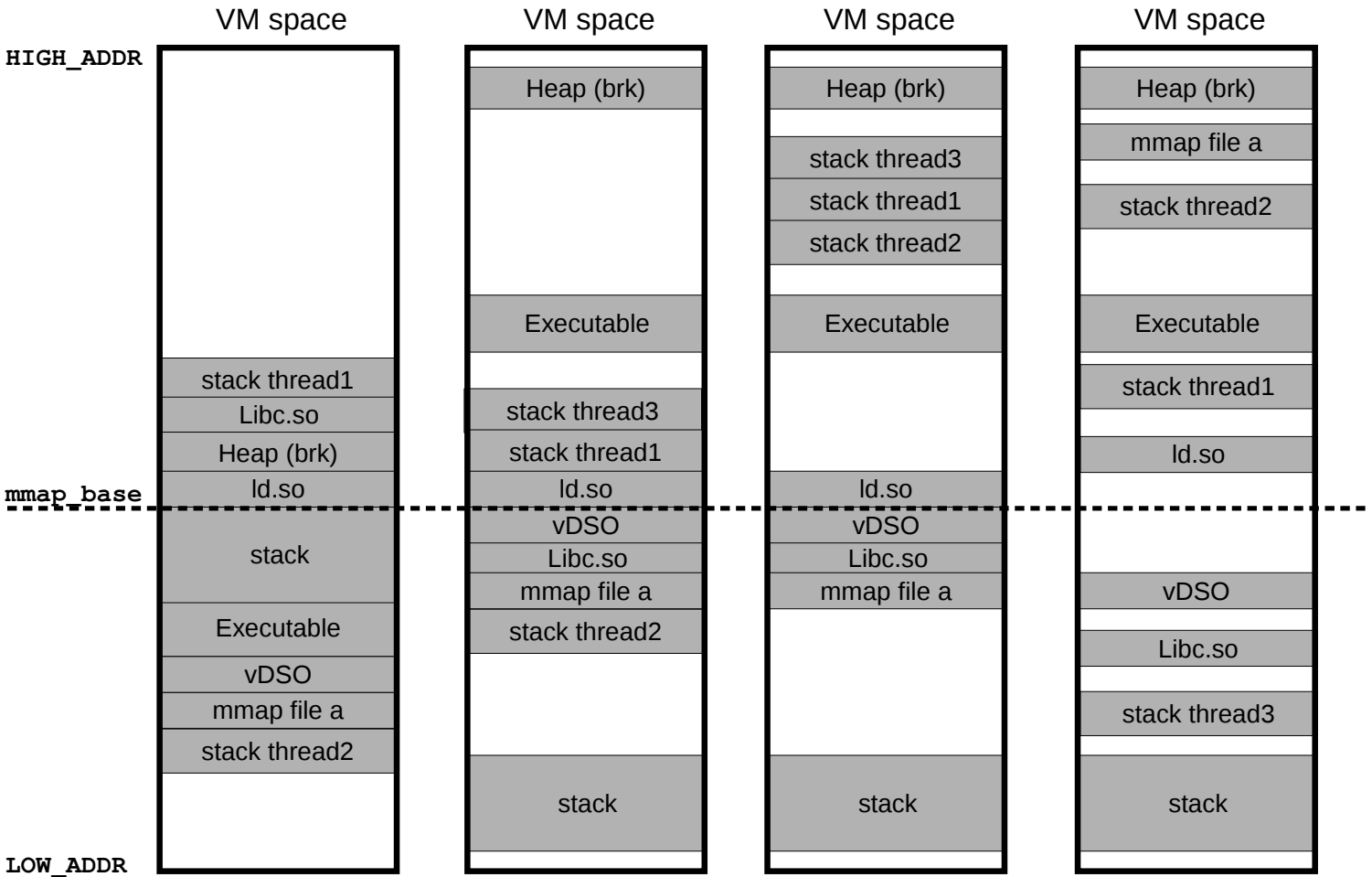
Linux ASLR Linux ASLR-NG

ASLR-NG: New randomisation forms

We have categorized and designed new randomization forms:

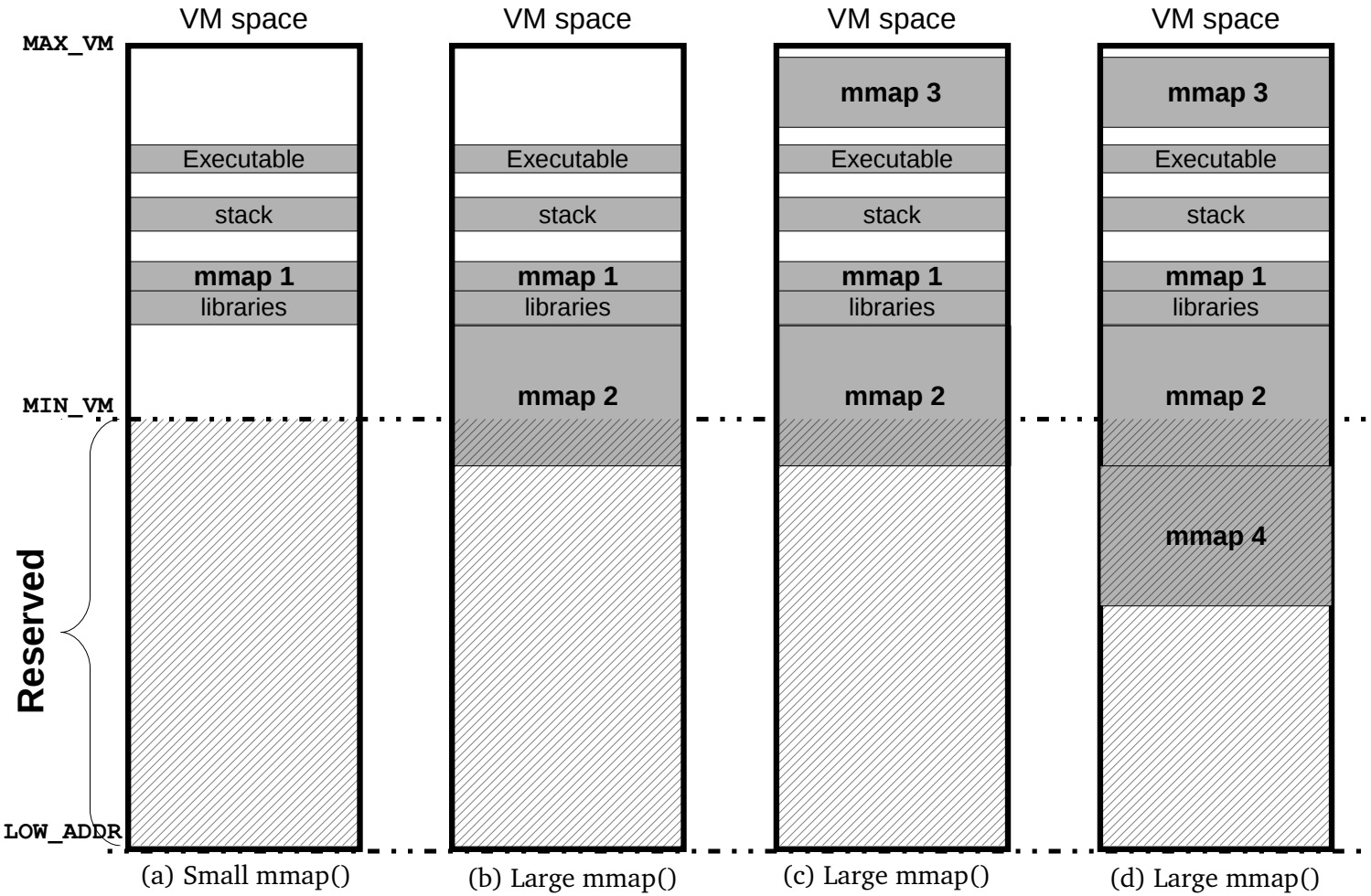
	Feature	Description
When	Per-boot	Every time the system is booted.
	Per-exec	Every time a new image is executed.
	Per-fork	Every time a new process is spawned. **new!** (inheritance weakness)
	Per-object	Every time a new object is created. **new!** (`MAP_RANDOM` flag)
What	Stack	Stack of the main process.
	LD	Dynamic linker/loader.
	Executable	Loadable segments (text, data, bss, ...).
	Heap	Old-fashioned dynamic memory of the process: `brk()`. **improved !**
	vDSO/VVAR	Objects exported by the kernel to the user space.
	Mmaps/libs	Objects allocated calling `mmap()`. **improved !**
How	Partial VM	A sub-range of the VM space is used to map the object.
	Full VM	The full VM space is used to map the object. **new!**
	Isolated-object	The object is randomised independently from any other. **new!**
	Sub-page	Page offset bits are randomised. **new!**
	Bit-slicing	Different slices of the address are randomised at different times. **Google!**
	Direction	Topdown/downtop search side used on a first-fit allocation strategy. **new!**
	Specific-zone	A base address and a direction where objects are allocated together. **new!**

ASLR-NG: Profile modes, dealing with page table sizes



- All modes have the same entropy. Only the correlation entropy is affected.
- The page table size can be tunned using the ASLR-NG profiles modes.

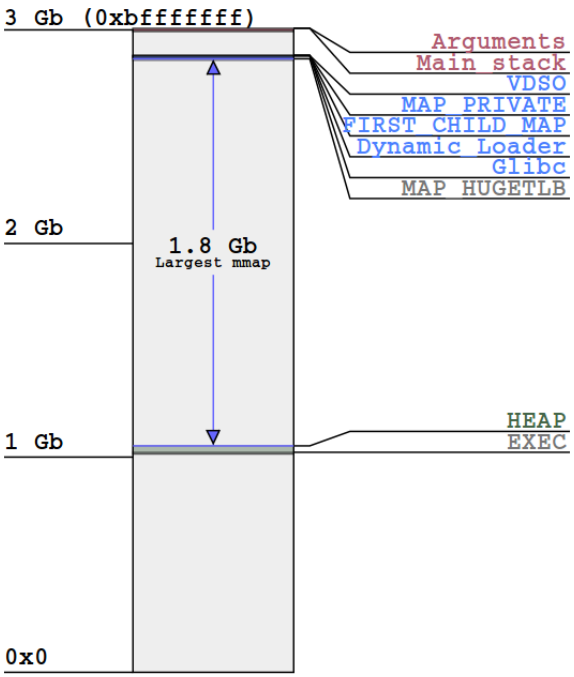
ASLR-NG: Solving the fragmentation issue (1/2)



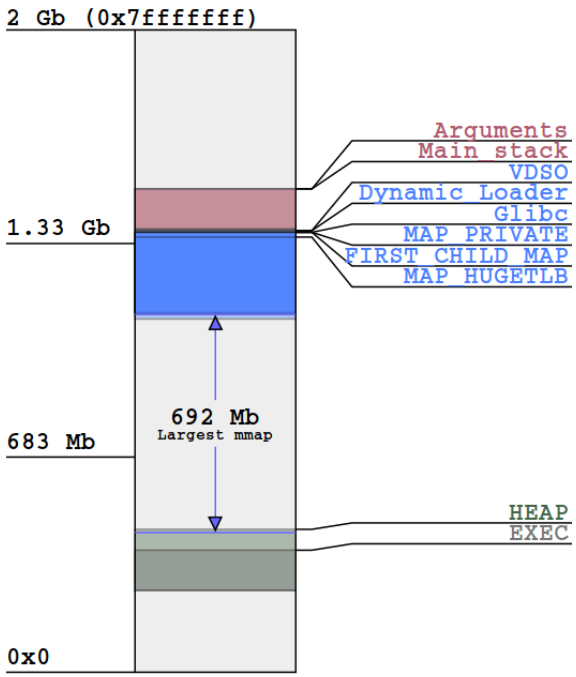
50% of the VA pre-reserved for large mmaps.

ASLR-NG: Solving the fragmentation issue (2/2)

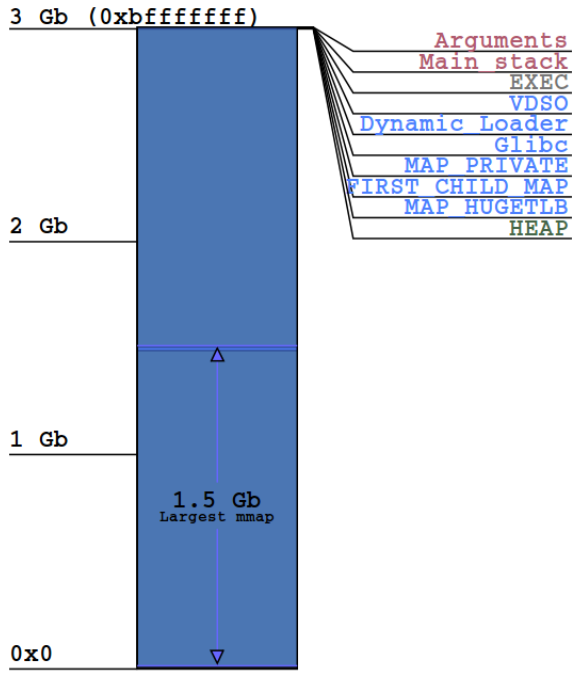
Linux (1.8 GB)



PaX (692 MB)



ASLR-NG (1.5 GB - 2.7 GB)



- Compatiblity is very important, some applications need free large portions of VMA.
- Requesting mmaps of 1 GB will fail in PaX (the price of having more entropy ?)
- Daniel Cashman ASLR proposal: Entropy vs fragmentation per system (same than PaX).
- ASLR-NG has both, maximum entropy with no fragmentation.

Linux vs PaX vs ASLR-NG 32-bits

Update
Trials x sec: 100

Object	RedHat 4.2.3 Entropy	RedHat 4.2.3 Time	PaX 3.14.21 Entropy	PaX 3.14.21 Time	ASLR-NG 4.5.0 Entropy	ASLR-NG 4.5.0 Time
Main_stack	19.0	1 hours	23.0	23 hours	27.6	23 days
HEAP	13.0	1 mins	23.4	1 days	27.6	23 days
Arguments	11.0	20 secs	27.0	15 days	31.6	1 year
EXEC	8.0	2 secs	15.0	5 mins	19.6	2 hours
Dynamic_Loader	8.0	2 secs	15.7	8 mins	19.6	2 hours
VDSO	8.0	2 secs	15.7	8 mins	19.6	2 hours
Glibc	8.0	2 secs	15.7	8 mins	19.6	2 hours
MAP_PRIVATE	8.0	2 secs	15.7	8 mins	19.6	2 hours
FIRST_CHILD_MAP	8.0	2 secs	15.7	8 mins	19.6	2 hours
MAP_HUGETLB	0.0	0 secs	5.7	0 secs	10.6	15 secs

- In 32-bit Linux ASLR is almost useless because of the low entropy.
- PaX is better than Linux but not enough (5 min the EXEC object).
- ASLR-NG provides the maximum possible absolute entropy.
- In Linux and PaX attackers can abuse of the low entropy of Hugepages (correlation weakness), but not in ASLR-NG.

Linux vs PaX vs ASLR-NG 64-bits

Update
Trials x sec: 100

	RedHat 4.2.3		PaX 3.14.21		ASLR-NG 4.5.0	
Object	**Entropy**	**Time**	**Entropy**	**Time**	**Entropy**	**Time**
Main_stack	30.0	124 days	35.0	10 year	43.0	$2 \cdot 10^3$ year
Dynamic_Loader	28.0	31 days	28.5	43 days	35.0	10 year
Glibc	28.0	31 days	28.5	43 days	35.0	10 year
MAP_PRIVATE	28.0	31 days	28.5	43 days	35.0	10 year
FIRST_CHILD_MAP	28.0	31 days	28.5	43 days	35.0	10 year
HEAP	28.0	31 days	35.0	10 year	43.0	$2 \cdot 10^3$ year
EXEC	28.0	31 days	27.0	15 days	35.0	10 year
Arguments	22.0	11 hours	39.0	174 year	47.0	$44 \cdot 10^3$ year
VDSO	21.4	7 hours	28.5	43 days	35.0	10 year
MAP_HUGETLB	19.0	1 hours	19.5	2 hours	26.0	7 days

- The executable in Linux has more entropy (twice) than PaX.
- PaX libraries and mmaps are non-uniform $\rightarrow$ effective entropy < 28.5.

  $\rightarrow$ Similar entropy than Linux
- In 64-bits, its effective against most attacks. (hugepages can be a threat in Linux and PaX).

Conclusions I

- We have presented 4 exploitable weaknesses which increases the probability of success for the attackers to bypass the ASLR.
- We have developed ASLRA, a tool which helped us to detected ASLR issues and measures its effectiveness.
  - In 32-bit PaX is better than Linux.
  - In 64-bit, Linux have twice entropy than PaX for the EXEC. and similar in libraries.
  - The paxtest tool measures the EXEC entropy incorrectly.
  - Linux provides a larger consecutive free space than PaX.
  - Hugepages not randomized in 32-bit Linux, ...
- A PoC of the correlation weakness (offset2lib) has been presented.
- We have contributed to the Linux ASLR:
  - CVE-2016-3672, Linux ASLR >= 4.x, Enable full randomization on i386 and X86_32, March 2016
  - CVE-2015-1593, Linux ASLR <= 3.19, Reduced stack entropy, Integer overflow, 2015
  - CVE (pending), Linux ASLR <= 4.0, AMD Linux ASLR weakness, Improper alignment, 2015
  - CVE (pending), Linux ASLR <= 3.18, Reduced mmap entropy, Improper mask manipulation, 2015

Conclusions II

- We designed and implement ASLR-NG, a new compatible and effective ASLR to address the weaknesses.
- The main features of **ASLR-NG** are:
  - Full VMA range to randomise objects $\longrightarrow$ maximum entropy.
  - A novel solution for reducing fragmentation, without reducing entropy.
  - Objects containing sensitive information are automatically isolated.
  - It provides a strong protection against absolute and correlation attacks.
  - New randomizations forms; Per-boot randomization, `MAP_RANDOM` ...
- ASLR-NG removes the 4 identified weaknesses providing both maximum entropy without fragmentation issues.
- ASLR-NG overcomes Linux and PaX ASLRs in all aspects.

Questions ?

* Hector Marco-Gisbert http://hmarco.org
* Ismael Ripoll Ripoll http://personales.upv.es/iripoll
* Cyber-security research group at http://cybersecurity.upv.es