Code sandboxing

Alpha Abdoulaye - Pierre Marsais

EPITA Systems/Security Laboratory (LSE)

July 14, 2017

Introduction

What do we want ?

- Limit usage of some resources such as system calls and shared object functions
- But not from the whole program (we trust our `libc.so`, `ld.so`, `...`)

Code sandboxing

- Needed when executing untrusted code on your machine.
- Allow or deny use of some “resources”
- Usually theses “resources” are accessed through syscalls
- We already have namespaces(7) and seccomp(2)

Are all needs fulfilled ?

There is no “ready-to-use” solution for:

- Function usage
- Library usage

Technical choices

Aim for:

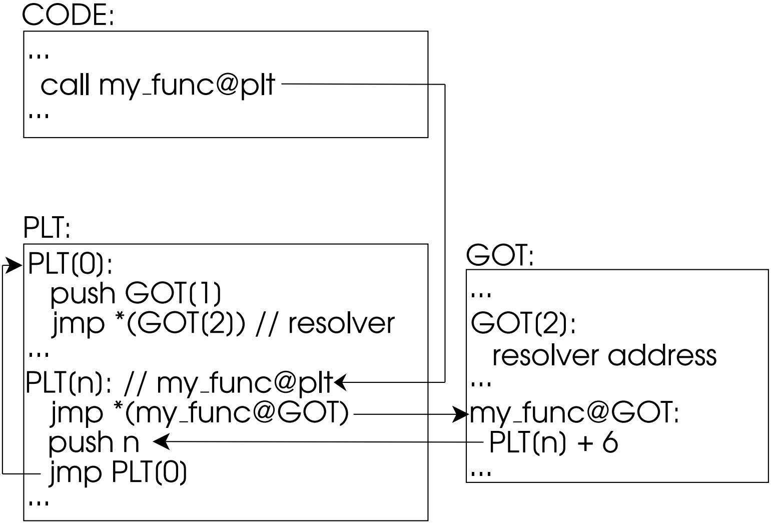
- Speed
- Reliability
- Security

Solutions

Solving the problem

- Trap at each function call
- Check if the call is righteous
- Continue as if nothing happened

How a library function is called



CODE:

```
...
 call my_func@plt
...
```

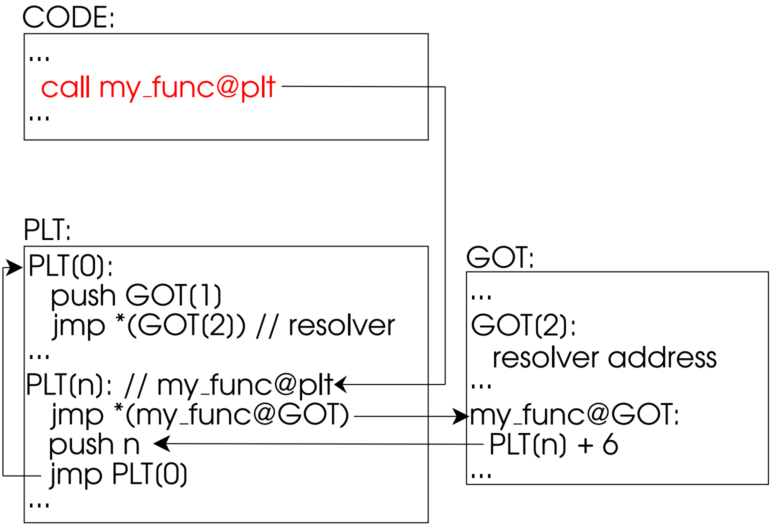
PLT:

```
PLT(0):
   push GOT(1)
   jmp *(GOT(2)) // resolver
...
PLT(n): // my_func@plt
   jmp *(my_func@GOT)
   push n
   jmp PLT(0)
...
```

GOT:

```
...
GOT(2):
  resolver address
...
my_func@GOT:
  PLT(n) + 6
...
```

How a library function is called



CODE:

```
...
 call my_func@plt
...
```

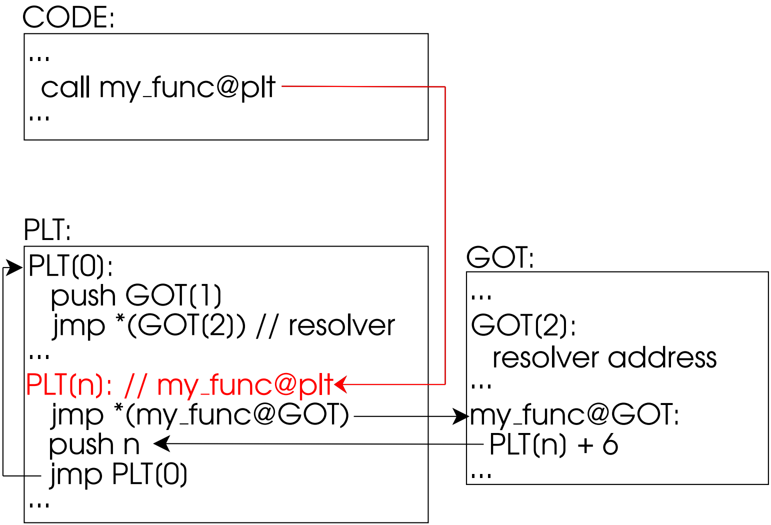
PLT:

```
PLT(0):
   push GOT(1)
   jmp *(GOT(2)) // resolver
...
PLT(n): // my_func@plt
   jmp *(my_func@GOT)
   push n
   jmp PLT(0)
...
```

GOT:

```
...
GOT(2):
   resolver address
...
my_func@GOT:
   PLT(n) + 6
...
```

How a library function is called



CODE:

```
...
 call my_func@plt
...
```

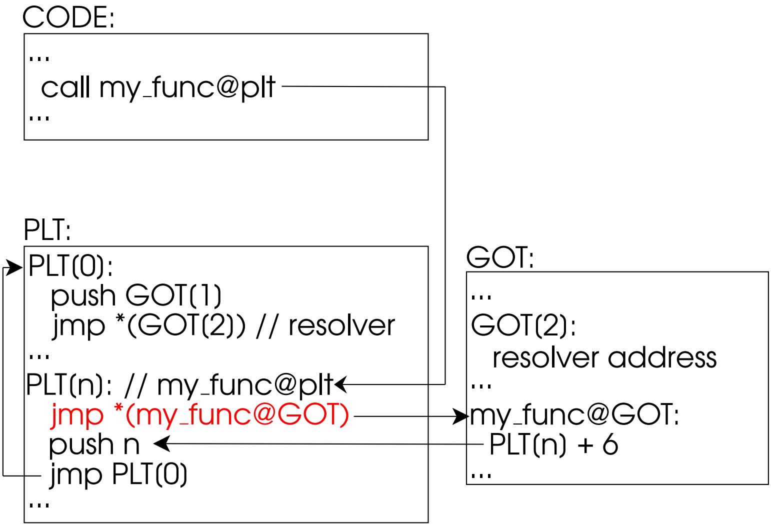
PLT:

```
PLT(0):
   push GOT(1)
   jmp *(GOT(2)) // resolver
...
PLT(n): // my_func@plt
   jmp *(my_func@GOT)
   push n
   jmp PLT(0)
...
```

GOT:

```
...
GOT(2):
   resolver address
...
my_func@GOT:
   PLT(n) + 6
...
```

How a library function is called



CODE:

```
...
 call my_func@plt
...
```

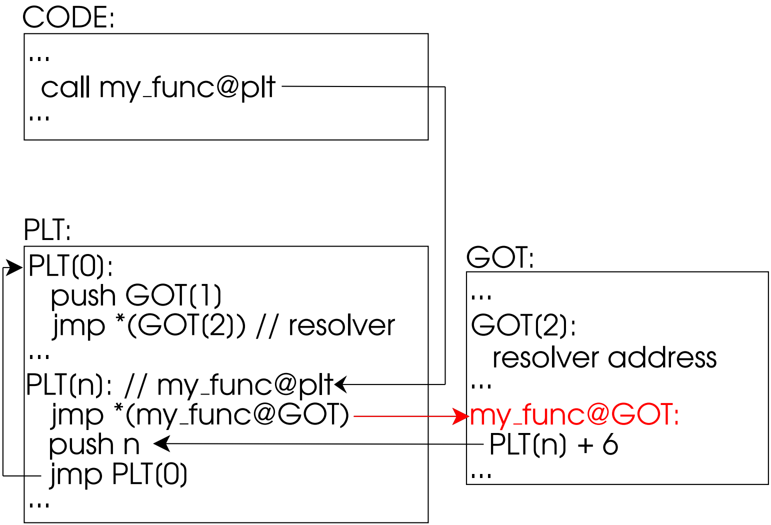
PLT:

```
PLT(0):
   push GOT(1)
   jmp *(GOT(2)) // resolver
...
PLT(n): // my_func@plt
   jmp *(my_func@GOT)
   push n
   jmp PLT(0)
...
```

GOT:

```
...
GOT(2):
   resolver address
...
my_func@GOT:
  PLT(n) + 6
...
```

How a library function is called



CODE:

```
...
 call my_func@plt
...
```

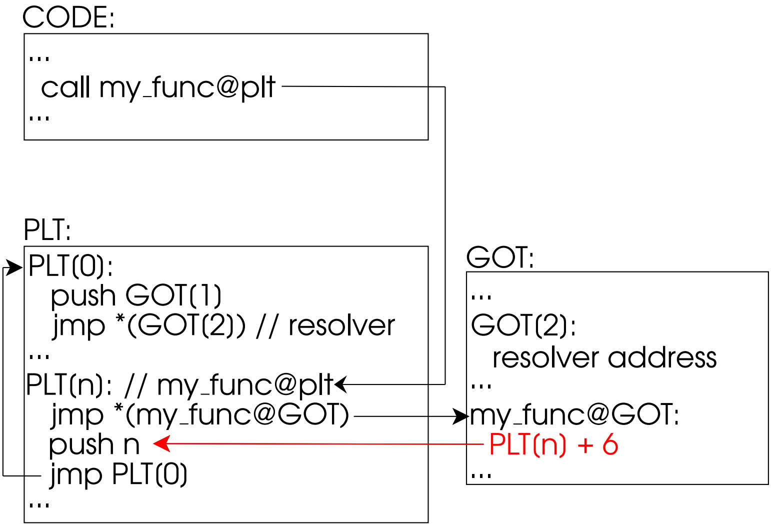
PLT:

```
PLT(0):
   push GOT(1)
   jmp *(GOT(2)) // resolver
...
PLT(n): // my_func@plt
   jmp *(my_func@GOT)
   push n
   jmp PLT(0)
...
```

GOT:

```
...
GOT(2):
   resolver address
...
my_func@GOT:
  PLT(n) + 6
...
```

How a library function is called



CODE:

```
...
 call my_func@plt
...
```

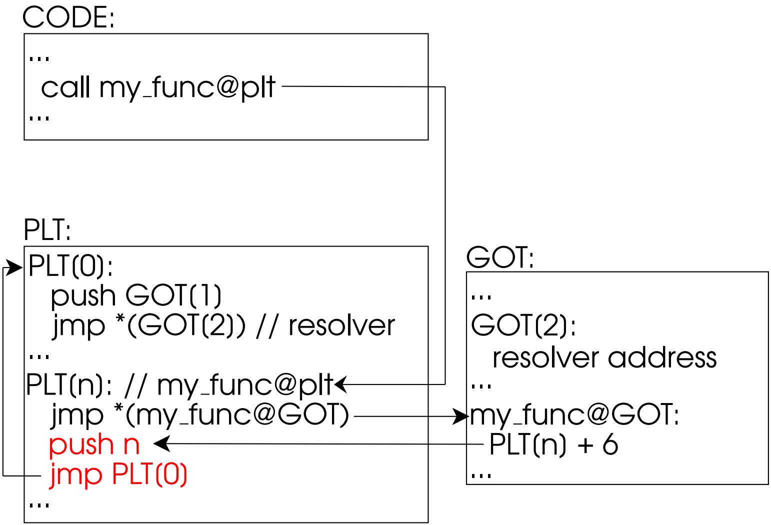
PLT:

```
PLT(0):
   push GOT(1)
   jmp *(GOT(2)) // resolver
...
PLT(n): // my_func@plt
   jmp *(my_func@GOT)
   push n
   jmp PLT(0)
...
```

GOT:

```
...
GOT(2):
   resolver address
...
my_func@GOT:
   PLT(n) + 6
...
```

How a library function is called



CODE:

```
...
 call my_func@plt
...
```

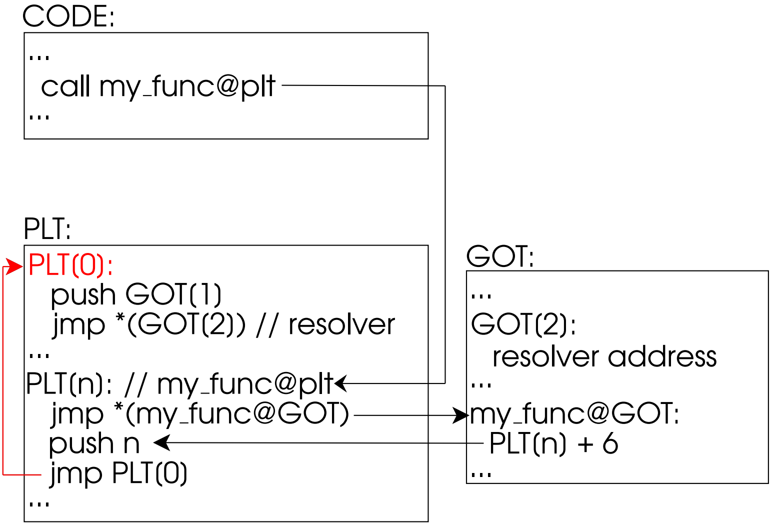
PLT:

```
PLT(0):
   push GOT(1)
   jmp *(GOT(2)) // resolver
...
PLT(n): // my_func@plt
   jmp *(my_func@GOT)
   push n
   jmp PLT(0)
...
```

GOT:

```
...
GOT(2):
   resolver address
...
my_func@GOT:
  PLT(n) + 6
...
```

How a library function is called



CODE:

```
...
 call my_func@plt
...
```

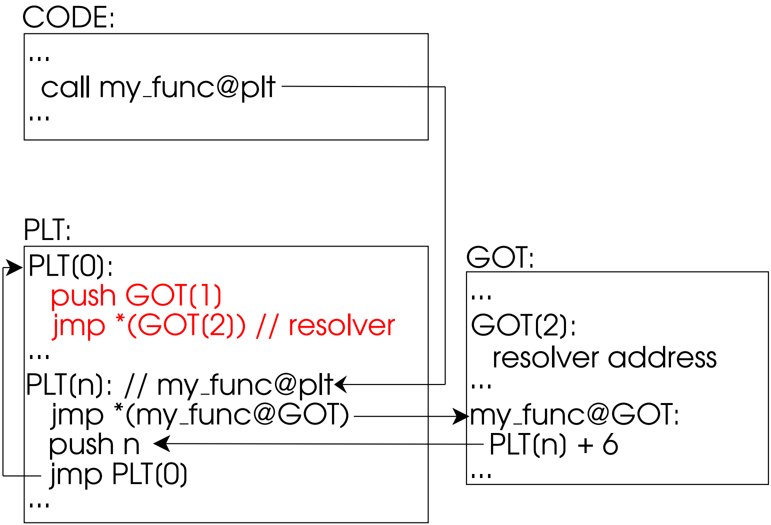
PLT:

```
PLT(0):
   push GOT(1)
   jmp *(GOT(2)) // resolver
...
PLT(n): // my_func@plt
   jmp *(my_func@GOT)
   push n
   jmp PLT(0)
...
```

GOT:

```
...
GOT(2):
   resolver address
...
my_func@GOT:
   PLT(n) + 6
...
```

How a library function is called



CODE:

```
...
 call my_func@plt
...
```

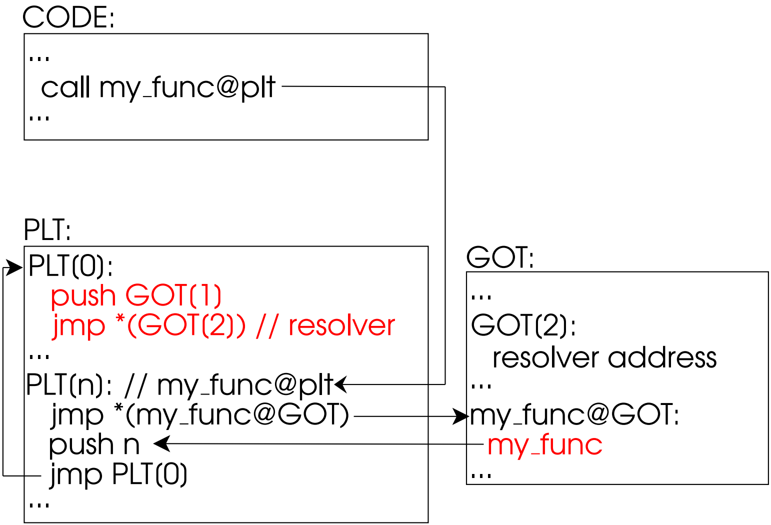
PLT:

```
PLT(0):
   push GOT(1)
   jmp *(GOT(2)) // resolver
...
PLT(n): // my_func@plt
   jmp *(my_func@GOT)
   push n
   jmp PLT(0)
...
```

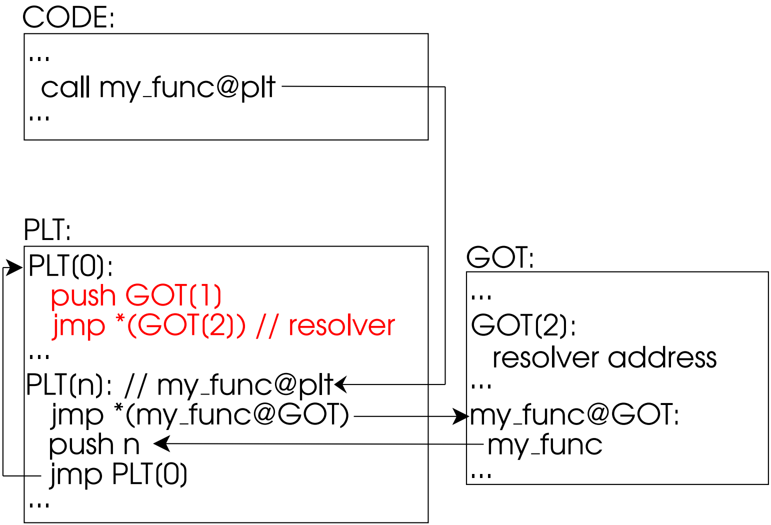
GOT:

```
...
GOT(2):
   resolver address
...
my_func@GOT:
  PLT(n) + 6
...
```

How a library function is called



How a library function is called



CODE:

```
...
 call my_func@plt
...
```

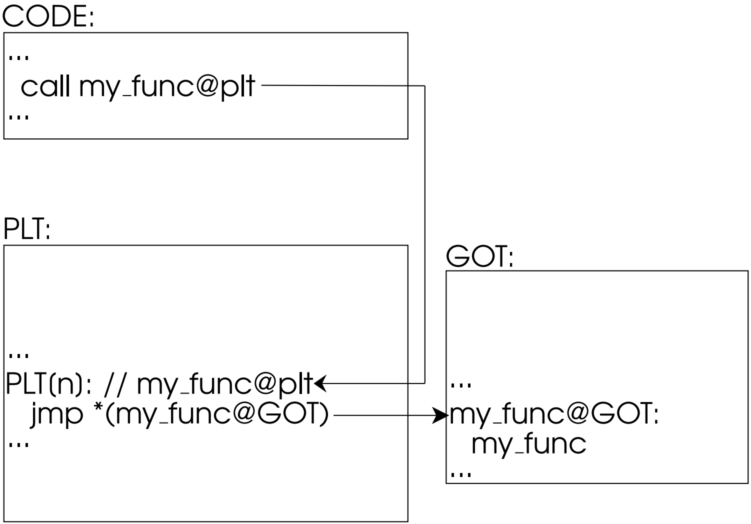
PLT:

```
PLT(0):
   push GOT(1)
   jmp *(GOT(2)) // resolver
...
PLT(n): // my_func@plt
   jmp *(my_func@GOT)
   push n
   jmp PLT(0)
...
```

GOT:

```
...
GOT(2):
   resolver address
...
my_func@GOT:
   my_func
...
```

Bound



CODE:

```
...
 call my_func@plt
...
```

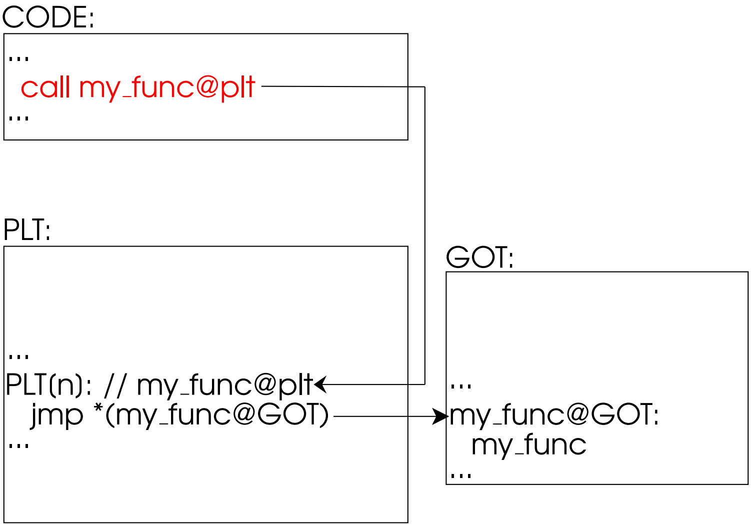
PLT:

```
...
PLT(n): // my_func@plt
  jmp *(my_func@GOT)
...
```

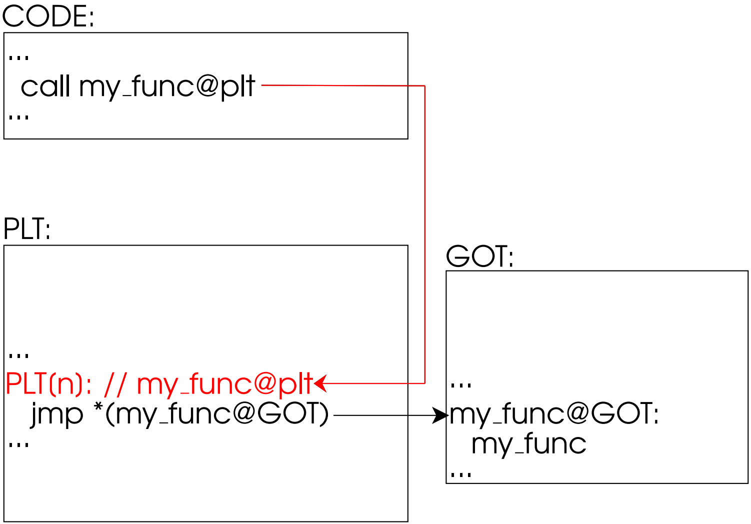
GOT:

```
...
my_func@GOT:
  my_func
...
```

Bound


CODE:
...
call my_func@plt
...
PLT:
...
PLT(n): // my_func@plt
jmp *(my_func@GOT)
...
GOT:
...
my_func@GOT:
my_func
...

Bound



CODE:

```
...
  call my_func@plt
...
```

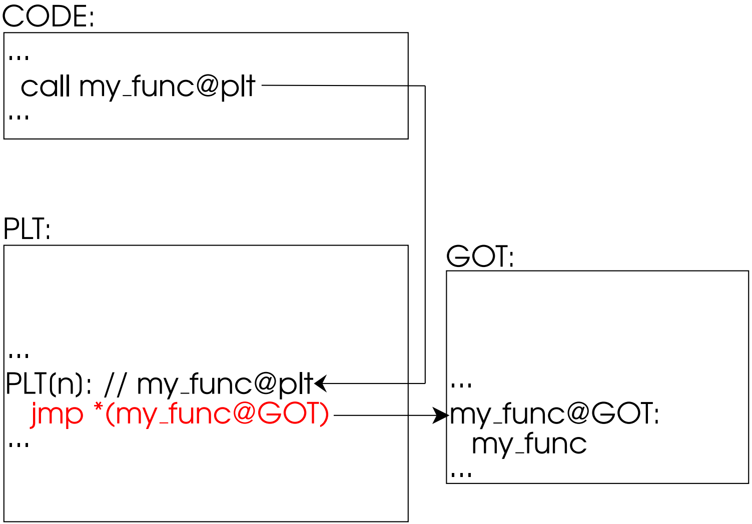
PLT:

```
...
PLT(n): // my_func@plt
   jmp *(my_func@GOT)
...
```

GOT:

```
...
my_func@GOT:
  my_func
...
```

Bound



CODE:

```
...
  call my_func@plt
...
```

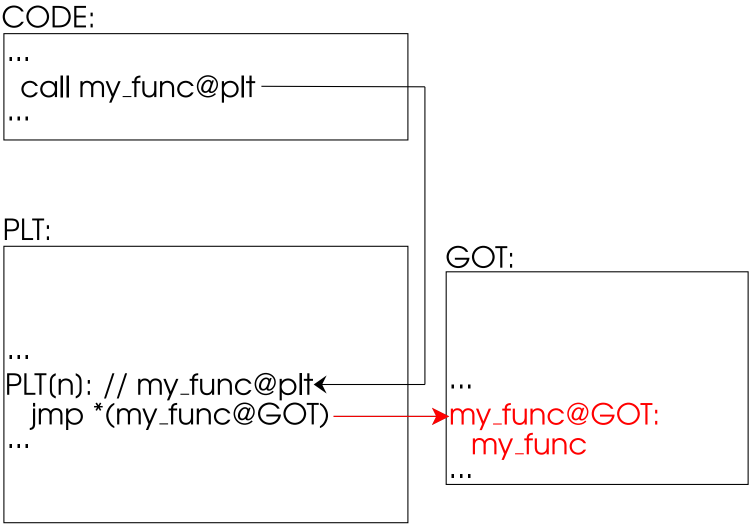
PLT:

```
...
PLT(n): // my_func@plt
   jmp *(my_func@GOT)
...
```

GOT:

```
...
my_func@GOT:
   my_func
...
```

Bound


CODE:
...
call my_func@plt
...
PLT:
...
PLT(n): // my_func@plt
jmp *(my_func@GOT)
...
GOT:
...
my_func@GOT:
my_func
...

So ?

We have two solutions:

- Disallow GOT reads of the sandboxed ELF
- Disallow code execution of executable mapping

Then handle the rights violation and check if the ressource access is allowed or not.

Solution analysis

GOT protection can be bypassed.
The correct solution would be unallowing execution of executable mappings.

Elf trickery

Goal

Can we solve our problem without privileged code ?

What do we need ?

- Change mapping rights
- Handle mapping violation

What can we use ?

- `ptrace(2)`
- `procfs(5)`

Is this enough ?

No:

- How to change mapping permissions from the tracer ?
- What about non-GOT data on GOT pages ?
- What about multithreaded programs ?

The first hack

How to change mappings from the tracer ?

- We can link an ELF to the sandboxed binary.
- We can use signal handlers in order to protect and unprotect the GOT.
- Use ELF constructors to setup everything.

The second hack

How to handle non-GOT data on GOT pages ?

- GOT doesn´t necessarily start and end at pages boundaries
- We can force this, with a custom linker script
- All we need is to customize the default linker script to align the GOT and export its size

We want reasonable performances

- `LD_BIND_NOW=1`
- Cache authorized GOT access

Limitations

We can't allow a lot of stuff for the sandboxed application:

- We currently need to link an object to the sandboxed application
- `mprotect` can't be used to `PROT_READ` the GOT
- `SIGSEGV` can't be handled
- Libraries addresses can be leaked

Problems

- Address space leaks
  - `/proc/self/*`
  - `auxv`
  - some syscalls
  - addresses on stack and structures
- Functions pointers in structures
- `dlopen`(3), `dlsym`(3)...

Shared object randomization

- Idea taken from OpenBSD
- If the user gets a libc address, and knows what libc is used, it can easily call any function
- The problem arise for any libs, but the libc is the more annoying for us
- We currently have a script to randomize the glibc
- Additional work needed for other libraries

Virtualization

We need to go deeper

- Extended Page Table
- Additional translation level
- Hardware assisted
- Solve multi-threading problem

Bareflank

- Lightweight
- Extendable
- “C++ in Kernel”
- Multi-platform

Track program behaviour

- `ptrace(2)`
- `/proc/[pid]/maps`
- `/proc/[pid]/pagemap`
- `linkmap, symbols, etc.`

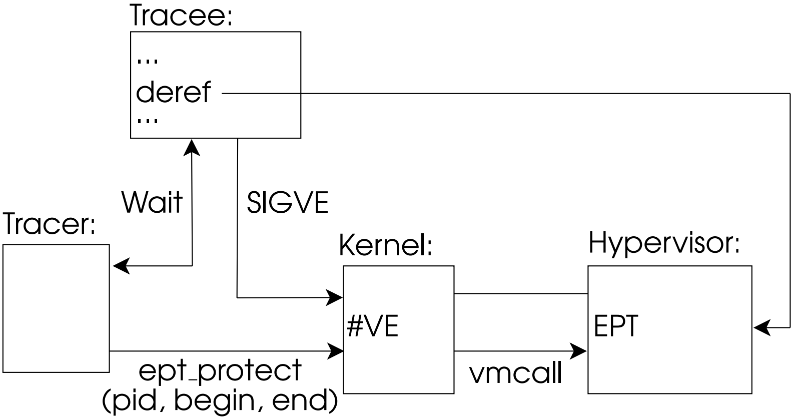
Report and Handle

- vmcall to report to hypervisor
- Virtual Machine Control Structure
  - VM State
  - Global Configuration
- VM Exits
- Enable EPT violation
- Convert to Virtualization Exception

Pull the strings

- Handle #VE
- Trap on protected code
- Protect executable and check
- Decide!!!!
- And so on. . .

Recap


Tracee:
...
deref
...
Wait
SIGVE
Tracer:
Kernel:
#VE
Hypervisor:
EPT
ept_protect
(pid, begin, end)
vmcall

Conclusion

Further work

- Be sure that our solution is foolproof
- handle multithreaded programs
- Work on performance
- What about statically linked ELFs ?
- ROP ?

Questions

Questions ?

Contact

- Alpha Abdoulaye - alpha@lse.epita.fr
- Pierre Marsais - pierre.marsais@lse.epita.fr