Linux Interactive Exploit Development with GDB and PEDA

Long Le
longld@vnsecurity.net

Workshop Setup (1)

- Virtual machine
  - VMWare / VirtualBox
  - Ubuntu 10.04+ Live CD ISO
  - Internet connection (NAT/Bridge)
- Install Ubuntu packages
  - Required packages
    ```
    $ sudo apt-get install nasm micro-inetd
    ```
  - Optional packages
    ```
    $ sudo apt-get install libc6-dbg vim ssh
    ```

Workshop Setup (2)

- PEDA tool
  - Download peda.tar.gz at: http://ropshell.com/peda/
  - Unpack to home directory
    ```
    $ tar zxvf peda.tar.gz
    ```
  - Create a ".gdbinit"
    ```
    $ echo "source ~/peda/peda.py" >> ~/.gdbinit
    ```
- Workshop exercises
  - Download bhus12-workshop.tar.gz at: http://ropshell.com/peda/
  - Unpack to home directory
    ```
    $ tar zxvf bhus12-workshop.tar.gz
    ```

Workshop Setup (3)

- Temporarily disable ASLR
  ```
  $ sudo sysctl -w kernel.randomize_va_space=0
  ```
- Allow ptrace processes
  ```
  $ sudo sysctl -w kernel.yama.ptrace_scope=0
  ```

Demo:
Sample Exploit Development session with GDB

GDB or not GDB?

- Standard debugger on *nix
- Not ExDev oriented
  - Lack of intuitive interface
  - Lack of smart context display
  - Lack of commands for ExDev
  - GDB scripting is weak
- Python GDB
  - Since GDB 7.0
  - Powerful scripting API (v7.2+)

PEDA Introduction

- Python Exploit Development Assistance for GDB
- Python GDB init script
  - GDB 7.x, Python2.6+
- Handy commands for exploit development
  - Self help manual
  - Auto-completion of commands, options
- Framework for writing custom commands

PEDA features

- Memory operations
- Debugging helpers
- Exploit helpers
- Utilities

Exploit Development with PEDA

Exploit Development Process

- Occupy EIP
- Find the offset(s)
- Determine the attack vector
- Build the exploit
- Test/debug the exploit

Occupied EIP, what next?

- Find the offset(s)
- Where is my buffer? Any register points to it?

Attack vector (1)

- Any exploit mitigation in place?
  - NX
  - ASLR
  - PIE
  - RELRO
  - CANARY

Attack vector(2)

- Find ways to code execution
  - ret2any: return to any executable, known place
    - stack
    - data / heap
    - text
    - library (libc)
    - code chunk (ROP)
  - control input buffer
    - stack pivoting

Build the exploit

- Payload
  - Shellcode
  - ret2any payload
- Wrapper
  - Exploit skeleton

Test and debug the exploit

- Check for limitation
  - Badchars
  - Buffer size
- Check for runtime affects
- Modify/correct the exploit

Demo & Practices

- Buffer overflow exploit
- Format string exploit
- PEDA commands explanation and usage

PEDA Commands

Prepare input buffer

- pattern create
```
pattern create 2000
pattern create 2000 input
```
- pset arg
```
pset arg '”A”*200'
pset arg 'cyclic_pattern(200)'
```
- pset env
```
pset env EGG 'cyclic_pattern(200)'
```

Context display

- Registers

```
context reg
```

- Code

```
context code
```

- Stack

```
context stack
```

Runtime info

- Virtual memory mapping

```
vmmap
vmmap binary / libc
vmmap 0xb7d88000
```

- Register / address

```
xinfo register eax
xinfo 0xb7d88000
```

- Stack / memory

```
telescope 40
telescope 0xb7d88000 40
```

Search for input buffer

- pattern offset
  ```
  pattern offset $pc
  ```
- pattern search
  ```
  pattern search
  ```

jmp/call search

- jmpcall

```
jmpcall
jmpcall eax
jmpcall esp libc
```

Generate shellcode/nopsled

- gennop

```
gennop 500
gennop 500 “\x90”
```

- shellcode

```
shellcode x86/linux exec
```

- assemble

```
assemble
```

Exploit wrapper

- skeleton

```
skeleton argv exploit.py
```

- Use with GDB

```
set exec-wrapper ./exploit.py
```

Memory search

- searchmem / find

```
find “/bin/sh” libc
find 0xdeadbeef all
find “..\x04\x08” 0x08048000 0x08049000
```

- refsearch

```
refsearch “/bin/sh”
refsearch 0xdeadbeef
```

- lookup address

```
lookup address stack libc
```

- lookup pointer

```
lookup pointer stack ld-2
```

ASM / ROP search

- asmsearch

```
asmsearch “int 0x80”
asmsearch “add esp, ?” libc
```

- ropsearch

```
ropsearch “pop eax”
ropsearch “xchg eax, esp” libc
```

- dumprop

```
dumprop
dumprop binary “pop”
```

- ropgadget

```
ropgadget
ropgadget libc
```

ELF headers / symbols

- elfheader / readelf

```
elfheader
elfheader .got
readelf libc .text
```

- elfsymbol

```
elfsymbol
elfsymbol printf
```

ret2plt / ROP payload

- payload

```
payload copybytes
payload copybytes target “/bin/sh”
payload copybytes 0x0804a010 offset
```

Other memory operations (1)

- dumpmem

```
dumpmem libc.mem libc
```

- loadmem

```
loadmem stack.mem 0xbffdf000
```

- cmpmem

```
cmpmem 0x08049000 0x0804a000 data.mem
```

- xormem

```
xormem 0x08049000 0x0804a000 “thekey”
```

- patch

```
patch $esp 0xdeadbeef
patch $eax “the long string”
pattern patch 0xdeadbeef 100
patch (multiple lines)
```

Other memory operations (2)

- strings

```
strings
strings binary 4
```

- hexdump

```
hexdump $sp 64
hexdump $sp /20
```

- hexprint

```
hexprint $sp 64
hexprint $sp /20
```

Other debugging helpers (1)

- pdisass
  ```
  pdisass $pc /20
  ```
- nearpc
  ```
  nearpc 20
  nearpc 0x08048484
  ```
- pltbreak
  ```
  pltbreak cpy
  ```
- deactive
  ```
  deactive setresuid
  deactive chdir
  ```
- unptrace
  ```
  unptrace
  ```

Other debugging helpers (2)

- stepuntil

```
stepuntil cmp
stepuntil xor
nextcall cpy
nextjmp
```

- tracecall / ftrace

```
tracecall
tracecall “cpy,printf”
tracecall “-puts,fflush”
```

- traceinst / itrace

```
traceinst 20
traceinst “cmp,xor”
```

Other debugging helpers (3)

- waitfor

```
waitfor
waitfor myprog -c
```

- snapshot

```
snapshot save
snapshot restore
```

- assemble

```
assemble $pc
> mov al, 0xb
> int 0x80
> end
```

- procinfo

```
procinfo
procinfo fd
```

Config options

- pshow

```
pshow
pshow option context
```

- pset option

```
pset option context “code,stack”
pset option badchars “\r\n”
```

- Edit lib/config.py for permanent changes

Python GDB scripting with PEDA (1)

- Global instances
  - pedacmd:
    - Interactive commands
    - Return nothing
    - e.g: `pedacmd.context_register()`
  - peda:
    - Backend functions that interact with GDB
    - Return values
    - e.g: `peda.getreg("eax")`
- Utilities
  - e.g: `to_int(), format_address()`

Python GDB scripting with PEDA (2)

- Getting help

```
pyhelp peda
pyhelp hex2str
```

- One-liner / interactive uses

```
gdb-peda$ python print peda.get_vmmap()

gdb-peda$ python
> status = peda.get_status()
> while status == “BREAKPOINT”:
>     peda.execute(“continue”)
> end
```

Python GDB scripting with PEDA (3)

- External scripts

```
# myscript.py
def myrun(size):
    argv = cyclic_pattern(size)
    peda.execute("set arg %s" % argv)
    peda.execute("run")

gdb-peda$ source myscript.py
gdb-peda$ python myrun(100)
```

Extending PEDA (1)

- PEDA structure
  - PEDA class
    - Interact with GDB
    - Backend functions
  - PEDACmd class
    - Interactive commands
  - Utilities
    - Config options
    - Common utils
    - External libraries

Extending PEDA (2)

- Special functions
  - `PEDA.execute()`
  - `PEDA.execute_redirect()`
  - `PEDACmd._is_running()`
  - `PEDACmd._missing_argument()`
  - `utils.execute_external_command()`
  - `utils.reset_cache()`

Extending PEDA (3)

- Writing new interactive command

```
class PEDACmd():
    ...
    def mycommand(self, *arg):
        """
        First line of docstring is the description of command
        Usage:
            MYNAME arg1 arg2
        """
        # get the arguments
        (arg1, arg2) = normalize_argv(arg, 2)
        # raise exception if missing argument
        if not arg1:
            self._missing_argument()
        # check if attached to running process
        if not self._is_running():
            return
        # use PEDA backend functions
        pid = peda.getpid()
        # generate output
        msg("My command: %d" % pid)

        return
```

Future plan

- More platforms
- ARM support
- Integration
  - IDA
  - libheap
  - libformat
  - CERT's exploitable

Thank you!