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

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

```
gdb-peda$ pyhelp peda
gdb-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
  - Available python libs (libheap, libformat, etc)
  - CERT's exploitable

Thank you!