GDB, Pwntools, and Other Tools

February 21st, 2019

Today's Goals

- Introduce several reverse engineering and exploitation tools and their purposes
  - These tools may be discussed further in future meetings
- Basic usage of GDB
- Understand several key features of pwntools

Announcements

- See https://sites.google.com/whitehatters.org/wcsc/announcements for CTFs we will be playing in this week

Reading Material

- GDB Documentation
  - The archer fish, GDB mascot
- Pwndbg - GDB Plug in
  - Features
- Pwntools Documentation
- Radare2
- Binary Ninja
- Angr

Introduction to the Tools

Below is an a list of tools for reverse engineering and exploit generation. Note that all of the tools can be found at https://sites.google.com/whitehatters.org/wcsc/new-members under the reverse engineering section.

Why these tools? Each of these tools serves a unique purpose, and have been found to be quite useful while working on CTF challenges. In addition, they provide a solid set of tools for increasing your understanding of the reversing material we cover during meetings.

Today we will be focusing on GDB and Pwntools, which I believe are the two most essential tools for beginning to develop your security knowledge in reverse engineering.

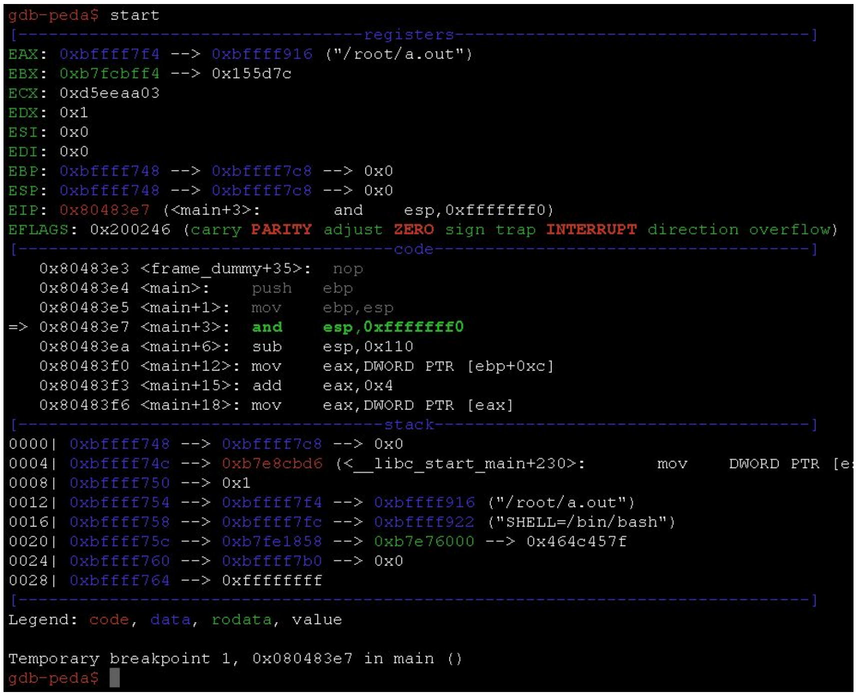
GDB—The GNU Project Debugger

The GNU Debugger, like most other debuggers, allows users to examine the execution of a program by setting breakpoints. The GNU debugger is meant to examine ELF executables, and provides a wide range of capabilities, including disassembling code segments and examining register values.

GDB has been seen several times already for these meetings, but today we will go over some of the most common commands.

Pwndbg and PEDA—GDB Extensions

These GDB extensions provide several common commands that make working with GDB easier. The most notable addition is the context provided after reaching each breakpoint (i.e. the debugger will print the stack, registers, and code every time a breakpoint is reached). The image below shows the context in PEDA.



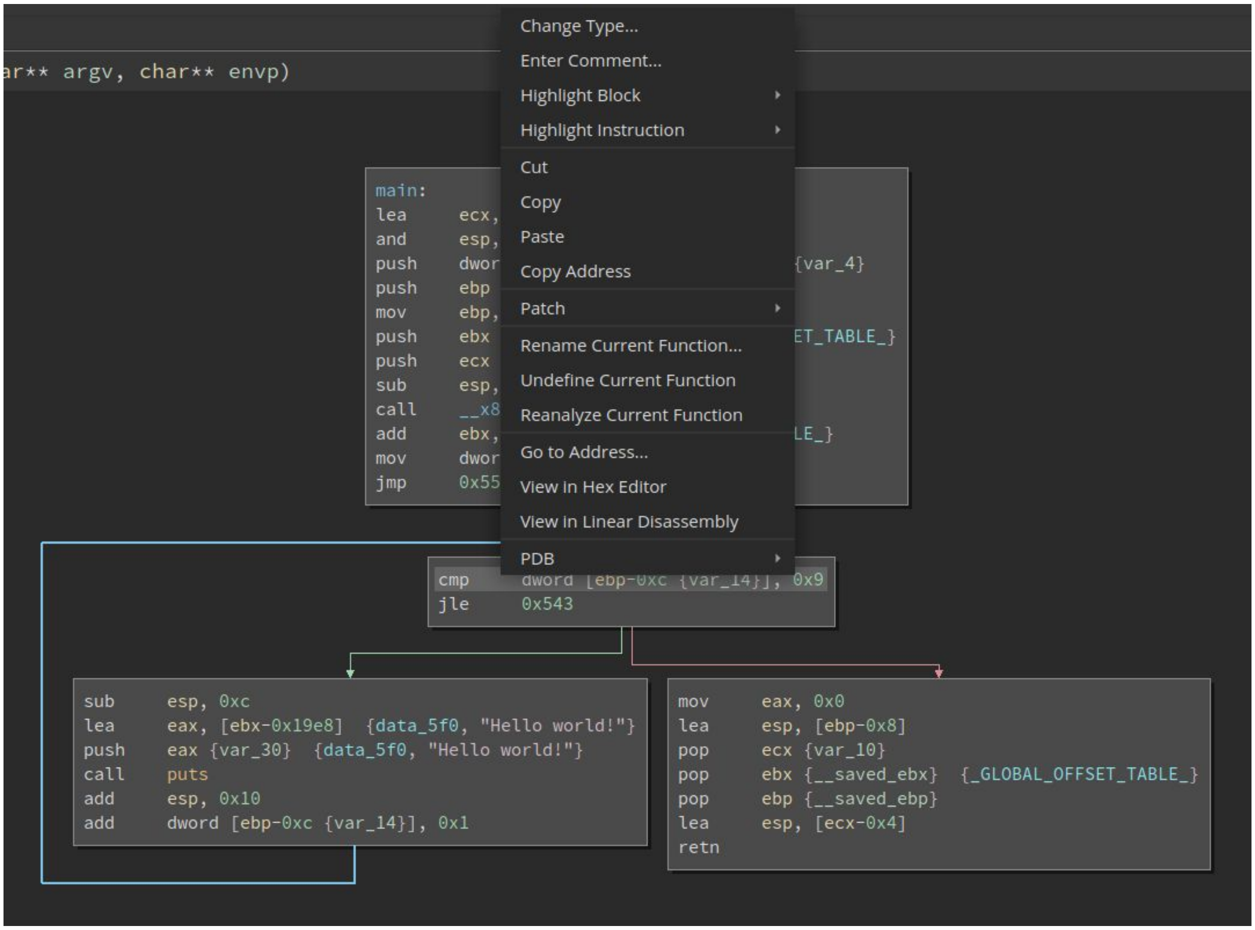
```
gdb-peda$ start
[----------------------------------registers-----------------------------------]
EAX: 0xbffff7f4 --> 0xbffff916 ("/root/a.out")
EBX: 0xb7fcbff4 --> 0x155d7c
ECX: 0xd5eeaa03
EDX: 0x1
ESI: 0x0
EDI: 0x0
EBP: 0xbffff748 --> 0xbffff7c8 --> 0x0
ESP: 0xbffff748 --> 0xbffff7c8 --> 0x0
EIP: 0x80483e7 (<main+3>:       and    esp,0xfffffff0)
EFLAGS: 0x200246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80483e3 <frame_dummy+35>:  nop
   0x80483e4 <main>:    push   ebp
   0x80483e5 <main+1>:  mov    ebp,esp
=> 0x80483e7 <main+3>:  and    esp,0xfffffff0
   0x80483ea <main+6>:  sub    esp,0x110
   0x80483f0 <main+12>: mov    eax,DWORD PTR [ebp+0xc]
   0x80483f3 <main+15>: add    eax,0x4
   0x80483f6 <main+18>: mov    eax,DWORD PTR [eax]
[------------------------------------stack-------------------------------------]
0000| 0xbffff748 --> 0xbffff7c8 --> 0x0
0004| 0xbffff74c --> 0xb7e8cbd6 (<__libc_start_main+230>:       mov    DWORD PTR [e
0008| 0xbffff750 --> 0x1
0012| 0xbffff754 --> 0xbffff7f4 --> 0xbffff916 ("/root/a.out")
0016| 0xbffff758 --> 0xbffff7fc --> 0xbffff922 ("SHELL=/bin/bash")
0020| 0xbffff75c --> 0xb7fe1858 --> 0xb7e76000 --> 0x464c457f
0024| 0xbffff760 --> 0xbffff7b0 --> 0x0
0028| 0xbffff764 --> 0xffffffff
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Temporary breakpoint 1, 0x080483e7 in main ()
gdb-peda$ 
```

When we discuss GDB in the following section, we will examine a few pwndbg commands.

Binary Ninja and IDA—Reversing Frameworks with a GUI

Binary Ninja and IDA provide a graphical interface for disassembling and reverse engineering programs. IDA is an industry standard, and is very popular, but has a hefty price range. Binary

Ninja is relatively new, and has some very nice features for a much more affordable price. Let's take a look at a program in Binary Ninja to get a grasp of the features.



```
ar** argv, char** envp)

main:
lea     ecx,
and     esp,
push    dwor                              {var_4}
push    ebp
mov     ebp,
push    ebx                               ET_TABLE_}
push    ecx
sub     esp,
call    __x8
add     ebx,                              LE_}
mov     dwor
jmp     0x55

cmp     dword [ebp-0xc {var_14}], 0x9
jle     0x543

sub     esp, 0xc
lea     eax, [ebx-0x19e8]  {data_5f0, "Hello world!"}
push    eax {var_30}  {data_5f0, "Hello world!"}
call    puts
add     esp, 0x10
add     dword [ebp-0xc {var_14}], 0x1

mov     eax, 0x0
lea     esp, [ebp-0x8]
pop     ecx {var_10}
pop     ebx {__saved_ebx}  {_GLOBAL_OFFSET_TABLE_}
pop     ebp {__saved_ebp}
lea     esp, [ecx-0x4]
retn
```

Radare2—Reversing Framework

Radare2 provides an entire framework for reverse engineering, including debugging, patching, and visual control flow graphs. Radare2 is one of the most flexible, free, and powerful reversing tools out there (take a look at their comparison chart), but it has a steep learning curve. Let's take a look at the same program we looked at in Binary Ninja with Radare2.

Use s main, aaa, agf to print the control flow graph for main. Show off the question mark command.

Pwntools—Python Library for Exploit Development

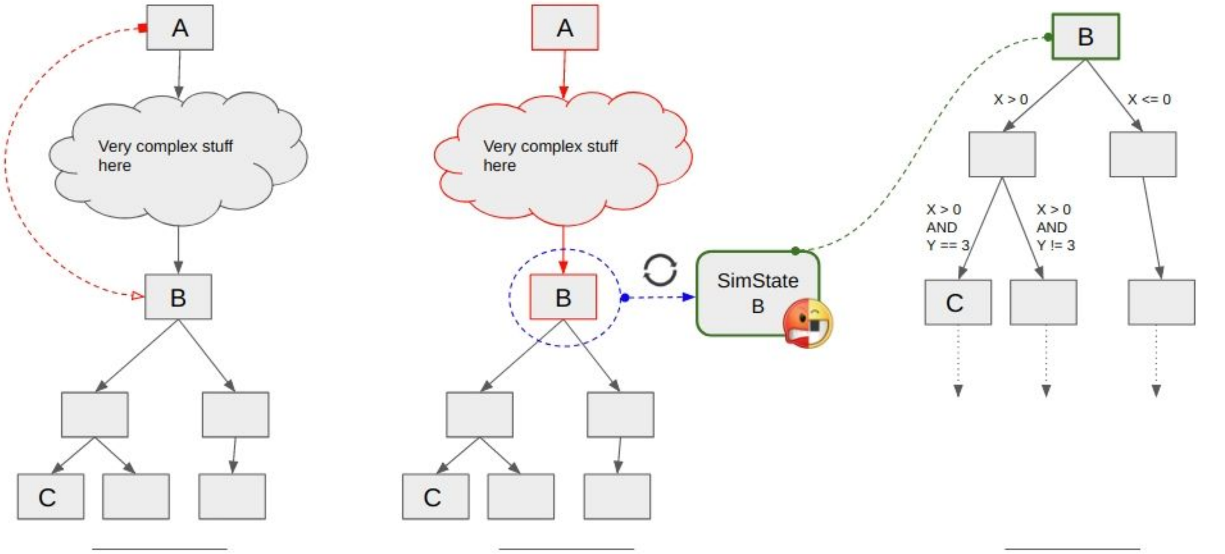
From their website, "pwntools is a CTF framework and exploit development library." Pwntools is written in Python, and provides many convenient functions for quickly solving CTF challenges. These functions include generating assembly and shellcode, ELF analysis such as symbol lookup, finding ROP gadgets, and printing values in little/big endian. We will take a look at several of these key features in the following section.

Angr—Concolic Analysis and Reversing Framework

What is concolic analysis? It is a combination of static (or concrete) and dynamic symbolic analysis.

Angr also provides several different useful tools, but symbolic execution is the most useful and unique capability it provides, and thus the focus for us.

The image below is a great tool for explaining at a high level what is going on in Angr.



Concrete execution from A to B

Synchronization of program's state in B inside angr

Symbolic execution from SimState B

In symbolic execution, each execution is a new state and the required values to reach that state are stored. In angr, to my knowledge, only branching states are stored to reduce the number of states. In the above diagram, you can see angr beginning to simulate state B. At each branch, angr creates an additional limit or test on the symbolic variable that must be true to reach that state. So, to reach state C, you must set X> 0 AND have y==3.

Where is this useful? Crackme challenges, which are comparable to busting licensing in software. In a crackme challenge, the goal is to determine some valid input that passes the validation checks. Angr could be used to simulate the execution and find what the value of the variable is when the correct code is reached. For example, say a crackme outputs “yay, you win” when xyz is entered. Angr can symbolically run to that point, and output the value needed to achieve that state.

If there is interest in the topic, a future meeting could discuss symbolic execution, and how angr does it, in more detail. It’s still a ast advancing research field in my opinion.

GDB

- Disassemble
- Break
- Step
- Stack (pwndbg)
- Info reg

Let’s take a look at a few GDB commands and what they do:

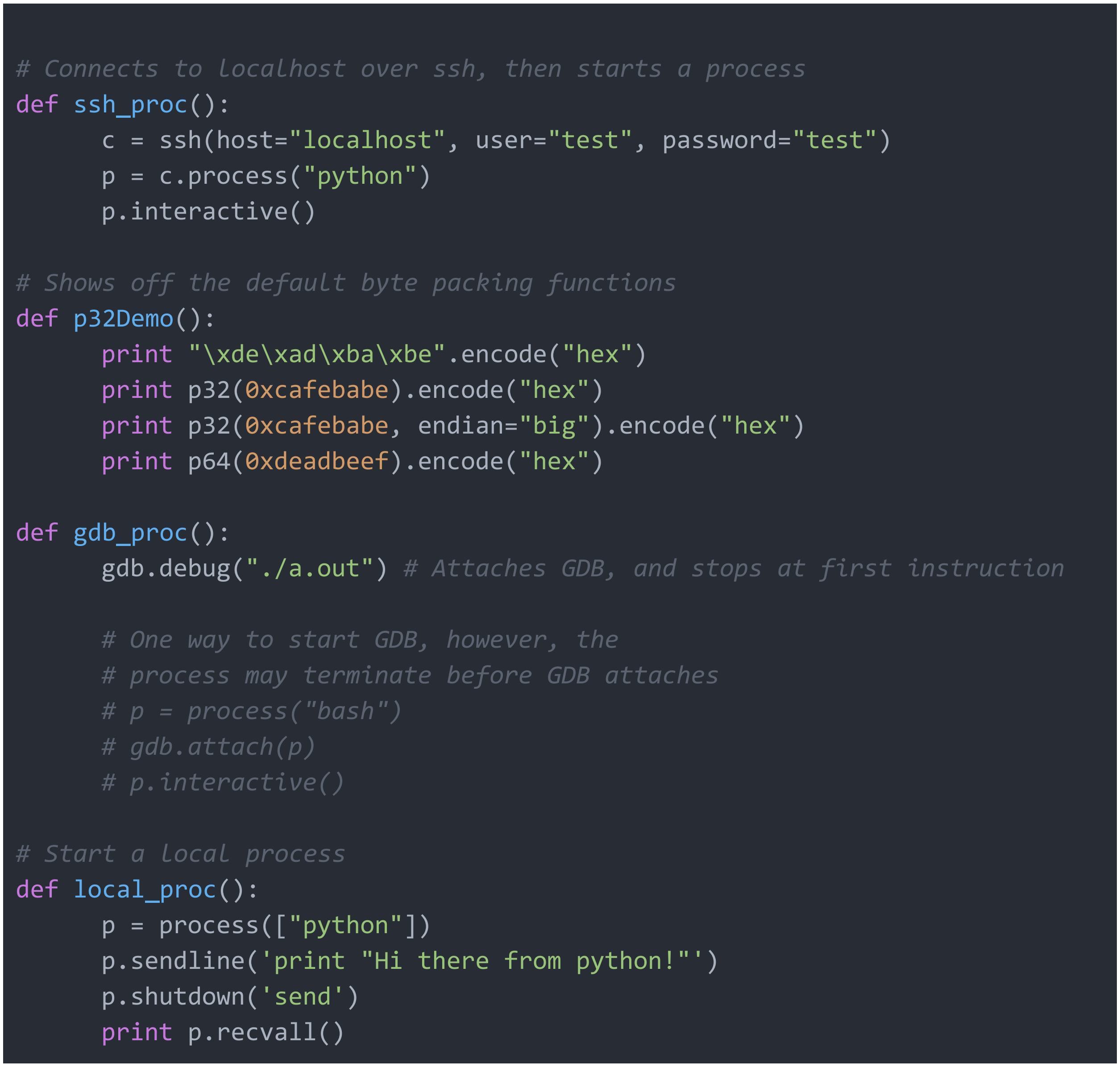
Command	**Shortcut**	**Purpose**
Help, help break	h, h disass	Prints a help message for the given command. No argument gives general help
break main, break 0x4000, break *main+0x4	b main	Sets a breakpoint at the specified address; note that an address or function name can be used
Run, run a b c	r	Run the program with arguments a, b, and c. Standard in/output redirection works like in bash (see below for a useful one)
run < <(python file)		Pipes the output of a command (such as python) as the stdin of the program
continue	c	Continue the program as normal until the next breakpoint
disassemble main, disassemble 0x4000, disassemble *main+0x4	disass main	Disassembles at the specified address and prints the resulting assembly code

step, step 10	s	Steps a specified number of instructions
next, next 10	n	Steps the specified number of instructions, stepping over function calls
finish	f	Finishes the current function, then breaks
info breakpoints, info reg, info reg eax	i b, i r, i r eax	Prints out the current breakpoints, register values, etc. See help i for more values
delete 1	del 1	Deletes breakpoint 1
stack	stack	Prints the stack (pwndbg command)
registers	regs	Prints the registers (pwndbg)
context	context	Prints the context; code, stack, registers, etc. (pwndbg)

Pwntools

Let’s take a look at some useful pwntools commands:

```
from pwn import *

# Open a "remote" connection
def show_local_remote():
      try:
            c = remote("localhost", 2000)
      except:
            return
      c.sendline("Please write 'me'")

      print c.recvuntil("me") # Wait until nc writes "me"
      print c.recvline()
      c.interactive() # Switch over to an interactive session

# Connects to localhost over ssh. Could be another remote server
def show_ssh():
      c = ssh(host="localhost", user="test", password="test")
      c.interactive()
```



```
# Connects to localhost over ssh, then starts a process
def ssh_proc():
      c = ssh(host="localhost", user="test", password="test")
      p = c.process("python")
      p.interactive()

# Shows off the default byte packing functions
def p32Demo():
      print "\xde\xad\xba\xbe".encode("hex")
      print p32(0xcafebabe).encode("hex")
      print p32(0xcafebabe, endian="big").encode("hex")
      print p64(0xdeadbeef).encode("hex")

def gdb_proc():
      gdb.debug("./a.out") # Attaches GDB, and stops at first instruction

      # One way to start GDB, however, the
      # process may terminate before GDB attaches
      # p = process("bash")
      # gdb.attach(p)
      # p.interactive()

# Start a local process
def local_proc():
      p = process(["python"])
      p.sendline('print "Hi there from python!"')
      p.shutdown('send')
      print p.recvall()
```