0x0000

a great fool in my life i have been
i have squandered 'til pallid and thin
hung my head in shame
and refused to take blame
for the darkness i know i've let win

j knapp

VulnCatcher:
Fun with Programmatic Debugging

atlas

atlas@r4780y.com

http://atlas.r4780y.com

0x0001 Who am I

- Scattered past in computing
- Insecurity Researcher
- Captain 1@stplace
- Father/Husband
- Curious fellow (sleepless too)

0x0100 Programmatic Debugging

- Debugging other processes from your (my) favorite language
- Accessing and Influencing CPU and Memory state of a process in a programmatic fashion
  - Logic and other language constructs

0x0101 Explosion

- Several key programmatic debuggers:
  - PyDBG – Pedram (part of Pai Mei)
  - Immunity Debugger – Immunity Sec
  - Vtrace – Invisigoth (Vivisect)
  - NoxDbg – Lin0xx (first Ruby debugger)

(This talk will focus on Vtrace)

0x0102 What can we do?

- Live Patching? Fun with Hex
  - LivePatch
- Live Dumping?
  - LiveOrganTransplant
- Process Grep?
  - Visi's memgrep
- Vampyre Jack SSHD
  - In progress by drb and myself

0x0103 What can we do?

- everything else that GDB or Olly can do, only better
- interactive python debugger
  - especially nice with *searchMemory()* and *traceme()*
  - *automate frame interpretation*
- *what do you **want** to do?*

0x0200 Vulnerabilities

- what can we do to encourage vulns to suddenly appear?
  - fuzzing on its own is so ghetto!
- rather, what can we watch/do to catch indications of vulnerability?

0x0300 Buffer Overflows?

- custom Breakpoints at key functions
- at break:
  - Stack-Analysis for Parameters
  - Buffer-Analysis for Size
- more empirical than static analysis

0x0301 vtrace attach

```
from vtrace import *
me = getTrace()
me.attach(<pid>)
me.addBreakpoint(MemcpyBreaker(me))
me.setMode("RunForever", True)
me.run()
```

0x0302 memcpy()

- memccpy()/mempcpy()/memmove()
  - check length of dest (%ESP + 0x4)
    - HEAP (dlmalloc), check length field immediately before the pointer to the dest
      - heapptr – 4
      - not always accurate.... copying partial chunks
    - Stack, check distance to RET
      - (%ebp + 4) – dest
        - oh, if only that simple...
  - compare with Copy Size (%ESP + 0xc)

0x0303 MemcpyBreaker

```
class MemcpyBreaker(BreakpointPublisher):
  def __init__(self):
    ...
  def notify(self, event, trace):
    eip=trace.getProgramCounter()
    esp=trace.getRegisterByName('esp')
    ebp=trace.getRegisterByName('ebp')
    copylen=trace.readMemoryFormat((esp + 0xc),AddrFmt)[0]
    retptr =trace.readMemoryFormat((esp + 0x0),AddrFmt)[0]
    dest   =trace.readMemoryFormat((esp + 0x4),AddrFmt)[0]
    src    =trace.readMemoryFormat((esp + 0x8),AddrFmt)[0]
    destlen = getBufferLen(dest)
    if (copylen >= destlen):
       self.publish(BOFException(...))
```

0x0400 EBP-FREE SUBS?

- some subs don't start new stack frames using %ebp
  - Windows Libraries
- trouble measuring stack buffer length

0x0401 EBP-FREE SUBS?

- some disassembly required...
- possible solutions:
  - Initial ESP Offset for Stack Allocation
  - Sub Epilog Analysis
    - ret $0x34
    - add $0x34, %esp
  - Sub Tracing for %esp Mods
    - 'til ret do us part
    - or jmp
  - OR.... Stack Backtrace for RET

0x0402 Stack Backtracing

- start at %ESP
- loop up the stack by 4 bytes
  - if the current 32-bit number is valid address (maps)
    - look for a “call” opcode immediately before the address
      - if so, is the target address valid?
        - is it a call to memcpy or a call to a jmp to memcpy
          - On Linux, does it target PLT?
- Once found, that location on the stack becomes RET
- Subtract the stack variable from the newly discovered RET location to find the length

0x0403 findRET()

```
def findRET(trace, stackptr = 0):
  cont = True
  stackptr = trace.getRegisterByName('esp')
  while cont:
    stackptr -= 4
    address = trace.readMemoryFormat(stackptr, AddrFmt)[0]
    mymap = trace.getMap(address))
    if mymap != None:   # valid address?
      buf = trace.readMemory(address-8, 8)
      for x in range(1,7):
        try:
          op = Opcode(buf[x:])
          if (op.off == 8-x and op.opcode[0] == 'c'):
            target = self.getOperandValue(op.dest)
            if trace.getMap(target) != None:
              # Possibly Check the Target of the call
              #  * Costly and not entirely accurate
              return address
```

(check the latest atlasutils for a much improved version)

0x0404 findNextHeap()

```
def findNextHeap(me, address):
   chain = getConnectedChain(me)
   for x in xrange(1,len(chain)):
      if chain[x] > address and chain[x-1] <= address:
         return chain[x]
```

0x0405 getConnectedChain()

- Finds HEAP memory map
- Searches for the first HEAP chunk
- Traverses the forward pointers
  - Keeps track and returns them as a list
- Works on Linux, not tried on Windows yet
- Look for it in the next release of atlasutils

0x0500 strcpy()/strncpy()

- strcpy – compare length of source and destination
  - dest pointer can be found at (%ESP + 0x4)
  - source pointer can be found at (%ESP + 0x8)

0x0501 strcpy()/strncpy()

- strncpy – compare length of copy (size_t) to destination
  - dest pointer can be found at (%ESP + 0x4)
  - size_t can be found at (%ESP + 0xc)

0x0502 strcat()/strncat()

- similar to strcpy/strncpy
- copies source and destination together
- difficult for coders to get right! (ie. often exploitable)
- best to look into logic surrounding strcat() limiting the size of both buffers

0x0600 printf()

- vfprinf covers printf and fprintf in Linux
- what's on the stack for format string?
  - %ESP + 0x8
- does it live in a likely spot?
  - Heap? Stack? .rodata?
- parse format string
  - are there “%” characters in it?

0x0601 sprintf()

- vsprintf covers sprintf in Linux
- what's on the stack for format string?
  - %ESP + 0x8
- does it live in a likely spot?
  - Heap? Stack? .rodata?
- parse format string
  - are there “%” characters in it?
  - how long of a string can we create?

0x0602 snprintf()

- vsnprintf covers snprintf in Linux
- what's on the stack for format string?
  - %ESP + 0x8
- does it live in a likely spot?
  - Heap? Stack? .rodata?
- parse format string
  - are there “%” characters in it?
  - how long will the format string allow?
  - how long **can** we write? (%ESP + 0xc)

0x0700 scanf/sscanf/fscanf

- parse format string
  - scanf's is located at %ESP+0x4
  - sscanf's and fscanf's are at %ESP + 0x8
- are there any “%s”?
- if so, where are we storing them?
  - must check each string
    - %45s against a buffer with 32 bytes

0x0800 gets()/fgets()

- lol.
- Just alert. Period.

0x0801 getc()/fgetc()

- loop for getc
- how big is the loop?
- simpler just to identify in disassembly and write up... analysis for which loop mechanism is used is more complex than just eye-balling it.

0x0900 memchr()/memrchr()

- check size_t against length of string as in memcpy
- may be used to look past a buffer as a potential target or source of data

0x0a00 rep stos/rep movs

- special case.
- need to disassemble code to hook these.
  - Set breakpoint one instruction before
  - stepi() to reach start of opcode
  - Check %ECX against buffer length

0x0b00 Format Strings

- used with printf/scanf families
- %c = 1 byte
- %* = * bytes (depends on the size)
- %#d = at **least** # bytes, possibly more!
- See man page for scanf or printf for more

0x0c00 Are there more?

- you tell me!
- programmatic debugging is fresh turf for new ideas.
- “The force runs strong in your family... Pass on what you have learned...”

0x0d00 choops

- hola y gracias amigos
  - Dios
  - jewel
  - bug
  - ringwraith
  - menace
  - 1@stplace
  - invisigoth and K