Stack Smashing as of Today

A State-of-the-Art Overview on Buffer Overflow Protections on linux_x86_64

<fritsch+blackhat@in.tum.de>
Hagen Fritsch − Technische Universität München

Black Hat Europe – Amsterdam, 17.04.2009

Me...

- Hagen Fritsch
- Informatics at Technische Universität München
  - Bachelor Thesis on hardware-virtualization Malware
  - Teaching in Networking and IT-Security classes
  - Specialisation in these fields, memory forensics & code verification
- Hacking at Home
  - Buffer overflows since pointers
  - Stack Smashing Contest @21C3
  - studivz-crawl
  - ...

Agenda

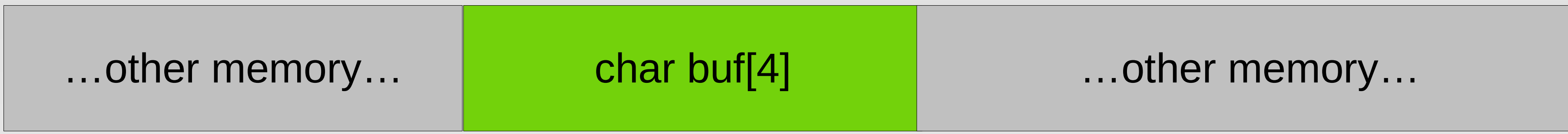
- Basic Principles, recap on buffer overflows
- Buffer Overflow Prevention
- Current Threat Mitigation Techniques
  - NX – Non-Executable Memory
  - Address Space Layout Randomization
  - Stack Smashing Protection / Stack Cookies
- Summary

Agenda

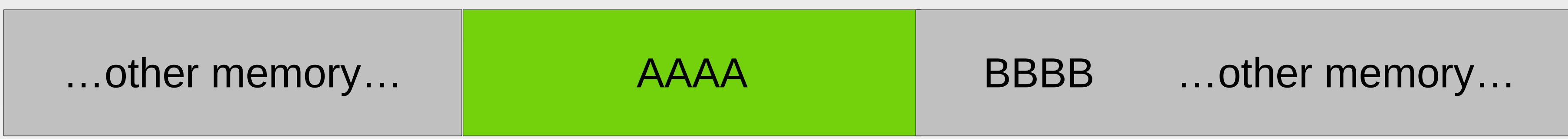
- **Basic Principles, recap on buffer overflows**
- Buffer Overflow Prevention
- Current Threat Mitigation Techniques
  - NX – Non-Executable Memory
  - Address Space Layout Randomization
  - Stack Smashing Protection / Stack Cookies
- Summary

Basics (Classic Buffer Overflows)

- `char buf[4];`




```
strcpy(buf, "AAAABBBB");
```



- Overwrites other memory, not belonging to buf

Basics (Classic Buffer Overflows)

- `char buf[4];`

…other memory…	char buf[4]	Int allow_root_access	…other memory…

```
strcpy(buf, ”AAAABBBB”);
```

…other memory…	*AAAA*	*BBBB*	…other memory…

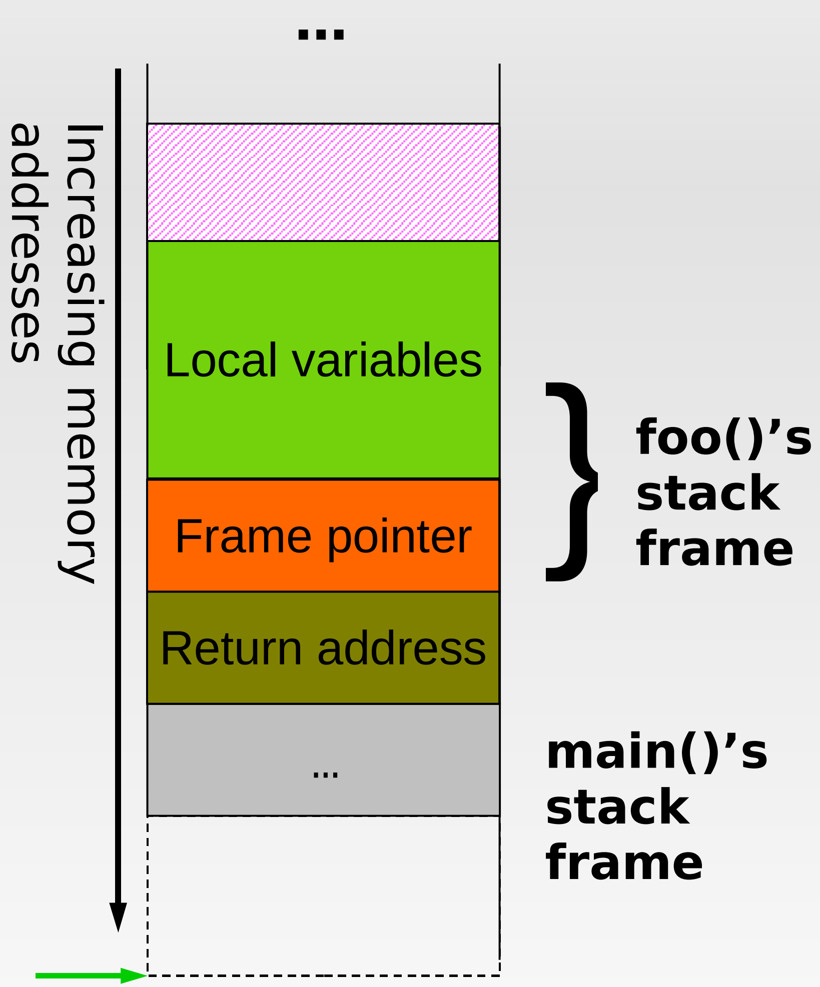
- Overwrites other memory,
  here: the allow_root_access flag

Classic Buffer Overflows (continued)

- Overwriting other variables' contents is bad enough (pointers)
- Bigger problem is:
  - Return addresses are stored on the stack

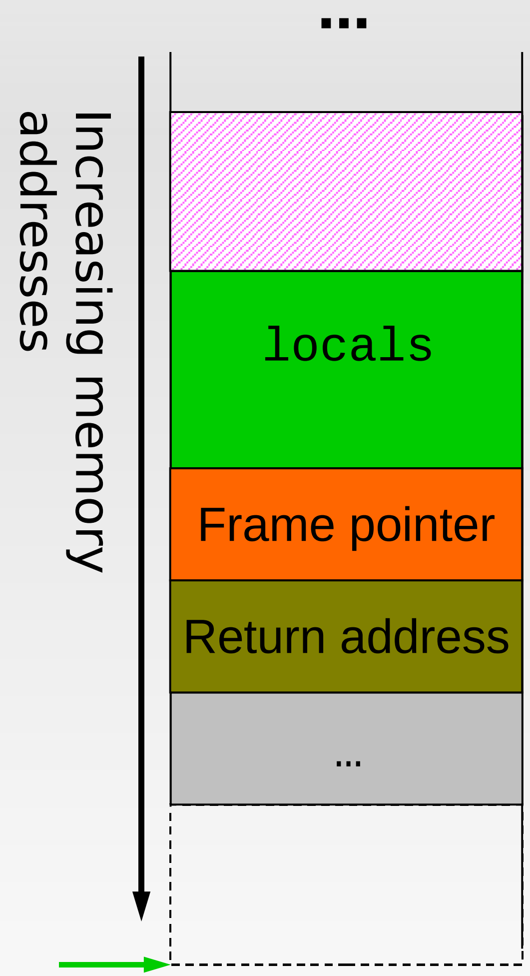
    e.g. in main():

```
     call foo
ret-addr: test %eax, %eax
```



Shellcode injection (still classic)

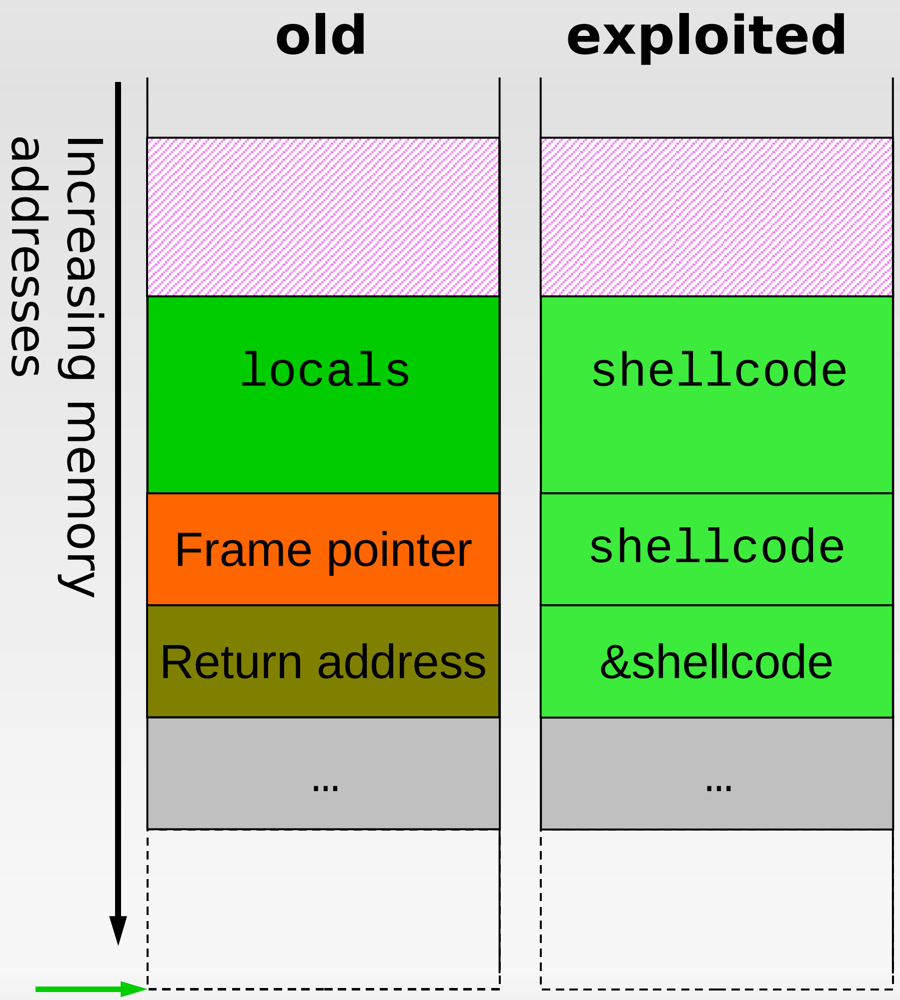
- Requirements
  - write arbitrary data into process address space
  - modify the return address (e.g. using a buffer overflow)
- Idea:
  - write own code on the stack and let it be executed



Shellcode injection (continued)

- Yes. How it works?
  - Put own code on the stack
  - Overwrite return address with shellcode's address
  - Function magically returns to and executes shellcode

c.f. "Smashing the stack for fun and profit", 1996



Agenda

- Basic Principles, recap on buffer overflows
- **Buffer Overflow Prevention**
- Current Threat Mitigation Techniques
  - NX – Non-Executable Memory
  - Address Space Layout Randomization
  - Stack Smashing Protection / Stack Cookies
- Summary

Buffer Overflow Prevention

- Some words on Prevention

  - Why do buffer overflows happen?
    - People make errors
    - Unsafe languages → Errors are easily made

  - How do we fix that?
    - Make people aware.
      - Did not work :'(
    - Make the language safe …?
    - Verify software …?

Buffer Overflow Prevention

- Bare pointers are evil
  - type-safe languages like Python, Ruby, Java etc. solve the problem
  - unfortunately noone will write an OS in Java (thanks god!)
- Dynamic approaches:
  - bounds-checking gcc
    - C is all about pointers and unbounded accesses
      - overhead sucks
  - Same goes for valgrind, although great tool
- Static verification – obviously fails
- Combined approaches
  - better, however still not practical

Agenda

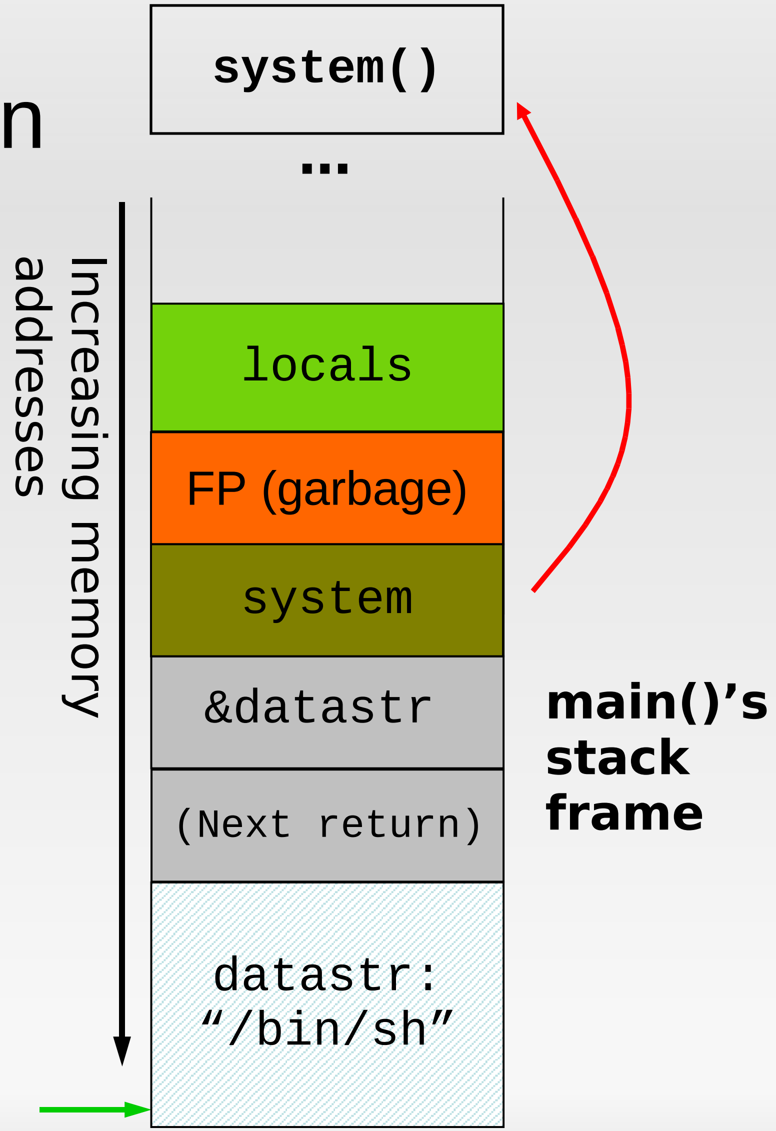
- Basic Principles, recap on buffer overflows
- Buffer Overflow Prevention
- **Current Threat Mitigation Techniques**
  - **NX – Non-Executable Memory**
  - Address Space Layout Randomization
  - Stack Smashing Protection / Stack Cookies
- Summary

NX — Preventing exploitation?

- Idea: make stack, heap etc. non executable
  - Code pages: r-x
  - Data pages (like stack, heap): rw-
  - Combination (r|-)wx MUST never exist!
- Effectively prevents *foreign code* execution
  - If applied (…correctly)
- The additional security came at some cost
  - Today: hardware-support, works like a charm

Circumventing NX: return into libc

- Who needs code execution at all if there are libraries?
- Goal: `system("/bin/sh")`
- ret-addr := &system
- arg1 := &datastr
- use `/////////…//////bin/sh` as "nops"



ret2libc first presented by SolarDesigner in 1997, and further elaborated by Rafal Wojtczuk
Phrack #58,4 has a summary on the techniques

Return into libc (x86_64)

- Calling conventions on x86:
  - ```
    push arg1
    call foo
    ```
- Calling conventions on x86_64
  - ```
    mov %rdi, arg1
    call foo
    ```
- Arguments in registers, thus not on the stack anymore

Return into libc (x86_64) (continued)

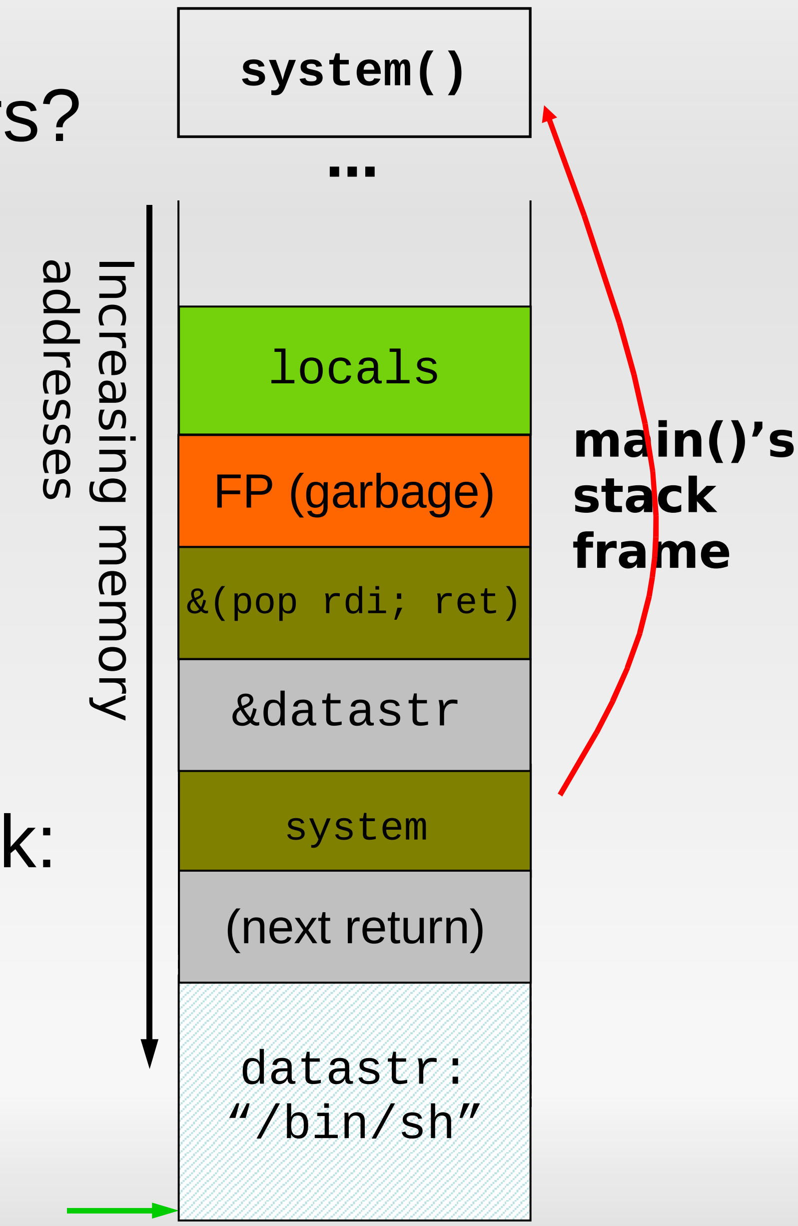
- How to get arguments into registers?
- Is there a function that does?

  pop %rdi

  ret

- Actually there is such a code-chunk:

  @__gconv+347 at the time of this writing



Ret code chunking

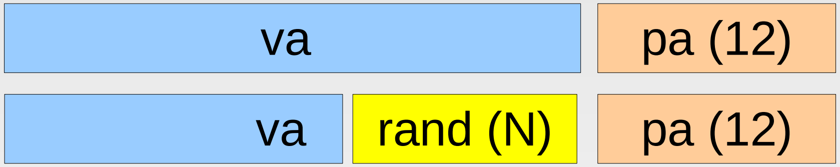
- Basically what we just did...
  - now: with arbitrary code fragments
- Idea:
  - Find parts of any shellcode’s instructions in libraries
  - Chunk them together by rets
- Conclusion: Non executable protection is no real drawback
  - Sorry, nothing new on NX. It’s pretty elaborated anyways.

Agenda

- Basic Principles, recap on buffer overflows
- Buffer Overflow Prevention
- **Current Threat Mitigation Techniques**
  - NX – Non-Executable Memory
  - **Address Space Layout Randomization**
  - Stack Smashing Protection / Stack Cookies
- Summary

ASLR (Address Space Layout Randomization)

- Observation: attacker needs to know precise addresses
  - make them unpredictable:
- OS randomizes each process’ address space
  - Stack, heap and libraries etc. are mapped to some “random address”
  - N bits of randomness



    - N actually varies depending on ASLR-implementation
    - Linux-Kernel:
      - Pages: 28 Bit (was only 8 bit on x86_32)
      - Stack: ~ 22 Bit, complicated obfuscation algorithm: 22 page_addr (2 of it discarded), 13 stack_top (4 of it discarded), 1 overlap with page_addr and another 7 lost likely because of PAGE_ALIGN

Circumventing ASLR

- 8 or 13 Bits is not much (28 bits suck though)
  - Use brute force … if feasible
  - because: fork(2) keeps randomization
    demonstrated by Shacham et. al (2004)
- execve(3) and a randomization bug
  - more to it soon
- Information leaks / partial RIP overwrites
  - cf. Phrack #59,9 “Bypassing PaX ASLR protection” (2002)
- Use loooong NOPs / plant hundreds of Megabytes of shellcode (Heap-Spraying)
  - won’t work in conjunction with NX

Circumventing ASLR (2)

- I liked ret2libc…
- … so are there executeable pages at static addresses despite ASLR?

```
# ldd /bin/cat
   linux-gate.so.1 => (0xffffe000)
   libc.so.6 => /lib/libc.so.6 (0xb7e19000)
   /lib/ld-linux.so.2 (0xb7f77000)
```

Circumventing ASLR (prior to 2.6.20)

```
# ldd /bin/cat
   linux-gate.so.1 => (0xffffe000)
   libc.so.6 => /lib/libc.so.6 (0xb7e19000)
   /lib/ld-linux.so.2 (0xb7f77000)
# ldd /bin/cat
   linux-gate.so.1 => (0xffffe000)
   libc.so.6 => /lib/libc.so.6 (0xb7d96000)
   /lib/ld-linux.so.2 (0xb7ef4000)
```

- Little flaw: linux-gate.so (Sorrow, 2008)
  - Syscall gateway
  - mapped into every process (at a fixed adress!)
  - borrowed code chunks :-)
    - jmp *%esp exists in linux-gate.so
    - and more stuff in case NX is in place (syscall gateway!)

Circumventing ASLR (after 2.6.20)

```
# ldd /bin/cat
   linux-gate.so.1 => (0xb7ff6000)
   libc.so.6 => /lib/libc.so.6 (0xb7e19000)
   /lib/ld-linux.so.2 (0xb7f77000)
# ldd /bin/cat
   linux-gate.so.1 => (0xb7ef3000)
   libc.so.6 => /lib/libc.so.6 (0xb7d96000)
   /lib/ld-linux.so.2 (0xb7ef4000)
```

- Little flaw: linux-gate.so
  - Fixed in 2.6.20 (February 2007)
- Anyways, how about x86_64?

Circumventing ASLR (on x86_64)

```
$ ldd /bin/cat
   linux-vdso.so.1 =>  (0x00007fffd4bff000)
   libc.so.6 => /lib/libc.so.6 (0x00007ff8cc66e000)
   /lib64/ld-linux-x86-64.so.2 (0x00007ff8cc9e0000)
$ ldd /bin/cat
   linux-vdso.so.1 =>  (0x00007fffc19ff000)
   libc.so.6 => /lib/libc.so.6 (0x00007f15b92c8000)
   /lib64/ld-linux-x86-64.so.2 (0x00007f15b963a000)
```

- Not promising at all

Circumventing ASLR (on x86_64)

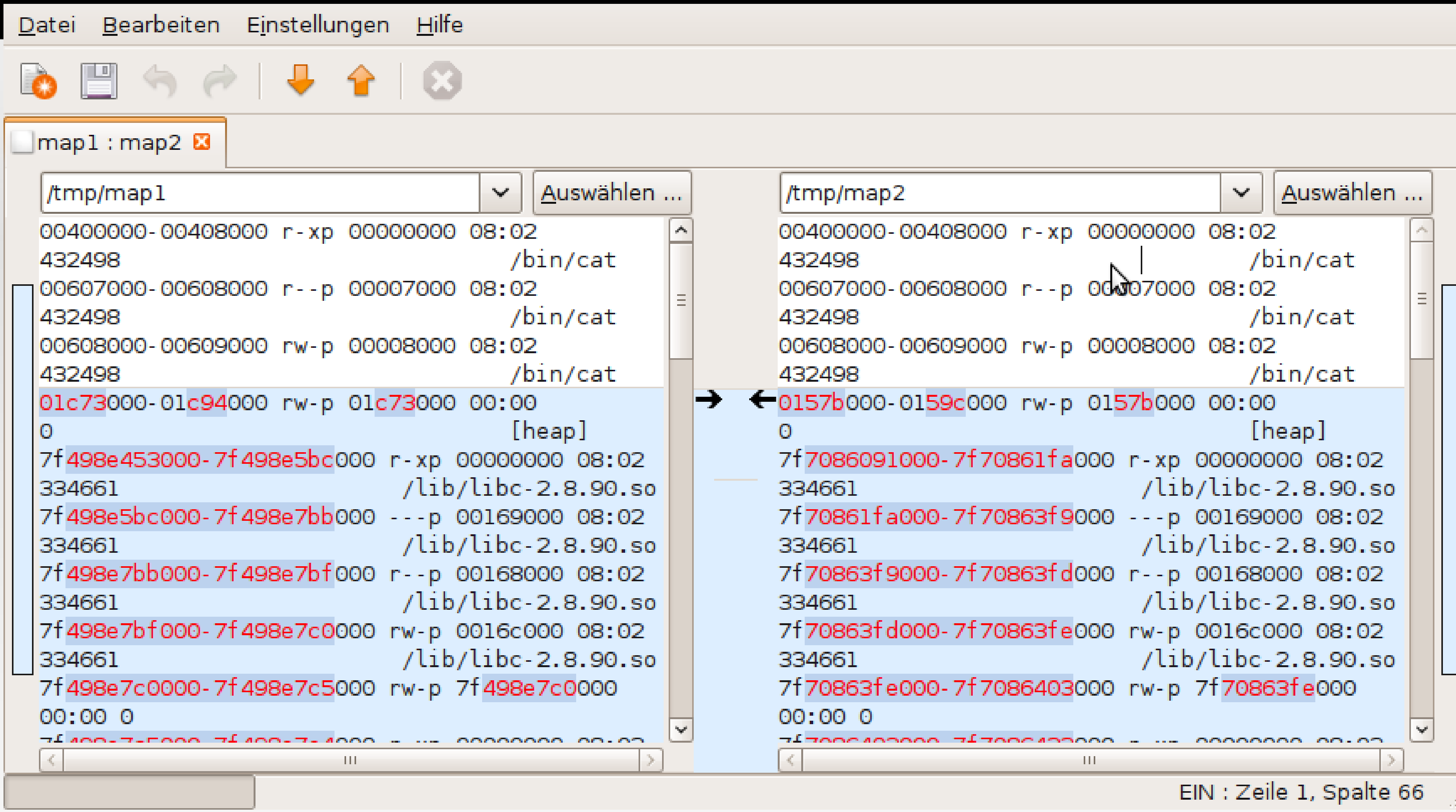
```
$ uname -rm
2.6.27-7-generic x86_64
$ cat /proc/self/maps
[...]
    7fff1f7ff000-    7fff1f800000 r-xp 7fff1f7ff000 00:00 0   [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

- Not promising at all? Except not quite!
- vsyscall kernel page at fixed address
  - `0xffffffffff600000`

vsyscall page

- Unfortunately nothing immediately obvious
  - No jmp/call *%rsp
  - Just a couple rare jmp/call *%register
  - Nearly no useful `ret` instructions
  - Work in progress...

Other static pages



- Code & Data-sections are not randomized
- Certainly contain interesting instructions
  - \x00 suck however…

A Linux Flaw

- Usage as in:

```
unsigned long arch_align_stack(unsigned long sp)
{
        If (!(current->personality & ADDR_NO_RANDOMIZE) &&
                                            randomize_va_space)
                sp -= get_random_int() % 8192;
        return sp & ~0xf;
}
```

- Randomness comes from here:

```
unsigned int get_random_int(void)
{
        /*
         * Use IP's RNG. It suits our purpose perfectly: it re-keys itself
         * every second, from the entropy pool (and thus creates a limited
         * drain on it), and uses halfMD4Transform within the second. We
         * also mix it with jiffies and the PID:
         */
        return secure_ip_id((__force __be32)(current->pid + jiffies));
}
```

The randomization Flaw (cont.)

- “every second” actually means: every 5 minutes
  - Not soo bad yet
- But something went wrong there s.t. secure_ip_id(x) is a PRF depending solely on x and the key
  - … which is only changed every 5 minutes
- Within that timeframe…
  - … get_random_int() depends solely on jiffies + pid

The randomization Flaw (cont. 2)

- State:
  - We don’t know jiffies or the secret key
  - We *know* the pid
  - We cannot compute the output of secure_ip_id()
    - (unless we could call it in kernel space…)
  - We don’t need to compute it

Exploiting the Flaw (same time)

- Impact 1:
  - within 4ms all launched processes with the *same pid* get the same randomization
  - launching a process using execve() *keeps* the pid
  - also for setuid-binaries

  - So lean back, read the randomization and run any service that helps you

Exploiting the Flaw (cont.)

- We cannot always start the vulnerable service
  - Someone else does this (e.g. init-scripts)
- However, we *can* recreate the conditions for secure_ip_id()
  - recall: `rand_int = secure_ip_id(pid + jiffies);`
  - Local attackers not only *know* the pid, they *control* it!
  - Assume now:
    - A service was just started.
    - We know *when* and its *pid*.

Recreating the *random* conditions

- As jiffies is a time-counter it constantly increases
- What happens if you fork() 32768 times?
  - Right, the pid wraps!
- small_jiffies + big_pid ⇔ bigger_jiffies + smaller_pid
  - Since jiffies increased, the pid needs to be decreased. That’s it!
- Caveats:
  - Jiffies has a granularity of 4ms
  - Userspace time-stamp /proc/%d/stat only 10ms
  - We need really good timing… and luck…
- Timeframe for attack: max. 32768 × 4ms ‣ 131s = 2m11s

Demo

- vuln_service is a forking network daemon (Google: server.c)
  - with an artificial vuln.
- Once exploit works without ASLR, all addresses just need the randomization-offset. So:
  - Acquire ~5-20 likely randomizations using a series of fork(), execve() and usleep()
- Try to exploit with each
  - One should succeed :-)

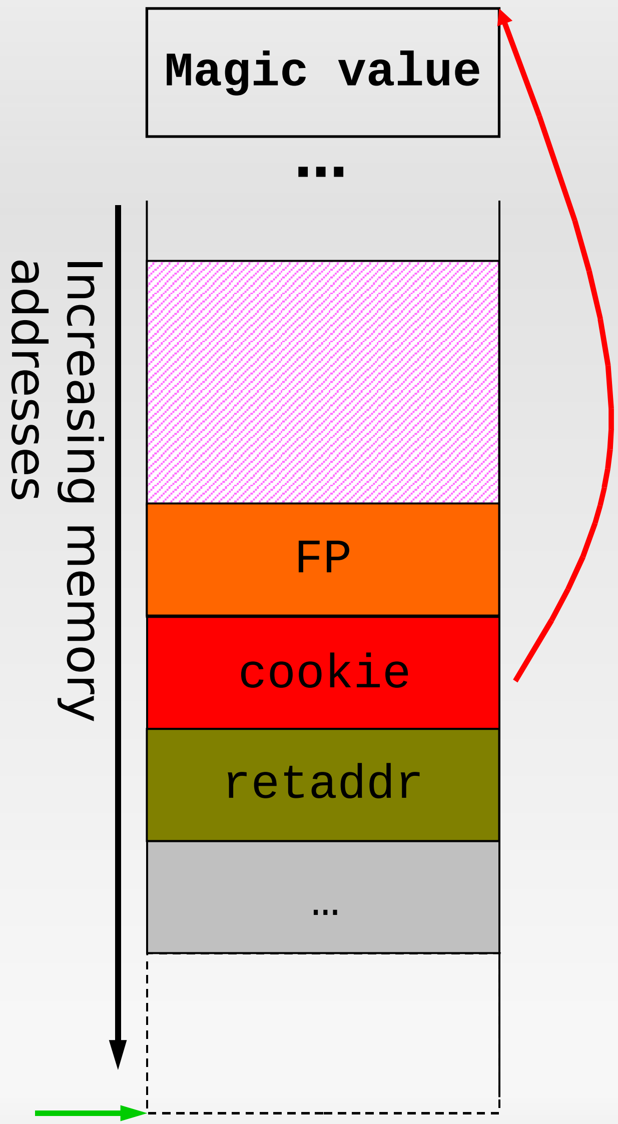
```
hagen@tuxinateur:~/blackhat/exploit$ ./guess_randomization `pidof server` | tee
out
        -6      stack randomization offset: 0x14b30b50 (page: 0x0f014b2f)
        -6      stack randomization offset: 0xa3fed010 (page: 0x00aa3fe9)
        -5      stack randomization offset: 0xa3fed000 (page: 0x00aa3fe9)
        -4      stack randomization offset: 0x456f3710 (page: 0x0f5456f0)
        -4      stack randomization offset: 0xedc47c60 (page: 0x0f6edc44)
        -3      stack randomization offset: 0xedc47c60 (page: 0x0f6edc44)
*       -2      stack randomization offset: 0x45153170 (page: 0x06145150)
*       -1      stack randomization offset: 0xca442460 (page: 0x09aca441)
*       -1      stack randomization offset: 0xcfa85aa0 (page: 0x04ccfa82)
*       0       stack randomization offset: 0xcfa85aa0 (page: 0x04ccfa82)
*       1       stack randomization offset: 0xd5ef4f10 (page: 0x0ead5ef1)
*       1       stack randomization offset: 0x1a7c57e0 (page: 0x0541a7c4)
*       2       stack randomization offset: 0x1a7c57e0 (page: 0x0541a7c4)
        3       stack randomization offset: 0xe28ec910 (page: 0x096e28eb)
        8       stack randomization offset: 0x9665b670 (page: 0x0c19665a)
```

Agenda

- Basic Principles, recap on buffer overflows
- Buffer Overflow Prevention
- **Current Threat Mitigation Techniques**
  - NX – Non-Executable Memory
  - Address Space Layout Randomization
  - **Stack Smashing Protection / Stack Cookies**
- Summary

Stack Smashing Protection (SSP)

- First introduced as stack cookies*
  - stored before the retaddr
  - it will be overwritten upon exploitation
- At function exit: If cookie does not match magic value:
  - Exit program
    (instead of returning to retaddr)



* later changed in gcc to xor cookie with framepointer
  now again cookie, but before FP (gcc 4.3.2 x86_64)

SSP (continued)

- Stack cookies in fact render most exploits impossible
  Not all of them! But at least stack-based buffer overflow attempts…
- …unless SSP protection is not in place
  - Only functions with char[] buffers > 4 byte are protected
- And: overwriting variables is still possible
  - Now think of pointers…
    - Object oriented code: vtables
  - Counter-countermeasure: variable reordering
    - ProPolice (IBM, ≈2005)
    - Aligning variables, seperating data and pointers

Getting around SSP

No need to give up too soon!

- A: don’t overwrite the cookie
  (e.g. pointer subterfuge)
- B: guess the cookie
  - Information leakage on the cookie
    - e.g. format string bugs (unlikely though)
  - side-channel timing guesses (Ben Hawkes, 2006)
- C: overwrite the master-cookie in TLS-area
  - Only possible for pointer-flaws like in (A)
  - ASLR is a bitch though.
- D: implementation flaws?

Stack canaries on Linux/glibc

- A closer look for case C – overwriting the master-cookie:
  - Canary stored in thread local area (TLS) at %fs:0x28
  - Initialized by ld.so
  - Located at a static location (assuming no ASLR)
  - a write64 can change it…
    - Less bits might be sufficient for certain cases

Stack canaries on Linux/glibc

- Implementation Flaws?
  - The pretty-much-static location is already bad
  - Let’s have a look at the source-code

Glibc dl-osinfo.h: canary initialisation

```
static inline uintptr_t __attribute__ ((always_inline))
_dl_setup_stack_chk_guard (void)
{
  uintptr_t ret;
#ifdef ENABLE_STACKGUARD_RANDOMIZE
  int fd = __open ("/dev/urandom", O_RDONLY);
  if (fd >= 0)
    {
      ssize_t reslen = __read (fd, &ret, sizeof (ret));
      __close (fd);
      if (reslen == (ssize_t) sizeof (ret))
        return ret;
    }
#endif
  ret = 0;
  unsigned char *p = (unsigned char *) &ret;
  p[sizeof (ret) - 1] = 255;
  p[sizeof (ret) - 2] = '\n';
  return ret;
}
```

setup_stack_chk_guard in practice

- ENABLE_STACKGUARD_RANDOMIZE is actually *off* on most architectures
  - Performance reasons
  - In this case canary defaults to 0xff0a000000000000
- Poor man's randomization hack by Jakub Jelinek: (applied at least in Fedora/Ubuntu)

```
def canary():
    __WORDSIZE = 64
    ret = 0xff0a000000000000
    ret ^= (rdtsc() &  0xffff) << 8
    ret ^= (%rsp   & 0x7ffff0) << (__WORDSIZE - 23)
    ret ^= (&errno & 0x7fff00) << (__WORDSIZE - 29)
    return ret
```

(Poor man's randomization hack)-attack

- Canary depends on
  - Address of errno
    - Static for a glibc (+ ASLR)
  - Address of the stack
    - Predictable (+ ASLR)
  - 16 lowest time-stamp bits
    - This actually sucks (16 bits are very kind though!)
- Now if we know those ASLR randomness…
  - … what remains are 16 bits of the TSC-value
  - write32 / write16 are sufficient to disable the protection
  - 16 bits are still in a possible brute force range…

Demo

- vuln_service is a forking network daemon (Google: server.c)
  - with an artificial vuln.
- Calculate canary for every 65536 possible timestamps
  - Exploit with each

    and have one succeed

```
hagen@tuxinateur:~/blackhat/sample$ python exp_server_canary.py
0000000 6e4b 650/serror connecting… <socket._socketobject object at 0x7ffff7f837
c0>
3

0000000 7035 490/serror connecting… <socket._socketobject object at 0x7ffff7f837
c0>
3

0000000 dc95 511/serror connecting… <socket._socketobject object at 0x7ffff7f837
c0>
3
```

Heap Overflows

- We haven't looked into them at all…
- However, they come down to write32s and there will always be those or similar vulnerabilities
  - Maybe not so much directly on heap
      - user-made data structures: linked lists, …
  - Pretty much exploitable with enough creativity
    - Sooooo many places in memory to ~~screw~~ write
    - Even NULL-pointer write32s are exploitable (c.f. Dowd's ridicilously crazy Flash exploit)
  - Minimize impact / harm they can do
    - No writeable and executable pages
    - Have ASLR in place (and update the kernel)

Agenda

- Basic Principles, recap on buffer overflows
- Buffer Overflow Prevention
- Current Threat Mitigation Techniques
  - NX – Non-Executable Memory
  - Address Space Layout Randomization
  - Stack Smashing Protection / Stack Cookies
- **Summary**
  - **Security is there – it’s just still a little broken**

Summary

Protection	Circumvention
NX	easy
ASLR	feasible
stack cookies	depends*
NX + ASLR	feasible*
NX + stack cookies	depends*
ASLR + stack cookies	hard*
NX + ASLR + stack cookies	hard*

* depends on environmental factors or certain code flaws

Thank you for your attention.

Any questions?