Binary encryption on UNIX

scut

Overview

- Any questions?
  - raise hand, will answer questions asap
- Duration: about one hour
  - 20 minutes: binary encryption
  - 20 minutes: ELF format
  - 10 minutes: demonstration
  - 10 minutes: questions :-)
- Documentation: http://www.team-teso.net/articles/18c3-encryption/

Binary encryption, wtf?

- object of interest: executeable files
- offers: protection against reverse engineering
- drawbacks: overhead, portability, pseudo-security
- history: DOS, Windows and shareware
- balance: real security vs. obscurity

Binary encryption on UNIX, why?

Commercial point of view

- need: protection of binary-only software (vs. OSS)
- need: commercial penetration testing

Attacker/Researcher point of view

- need: anti-forensics for cracker tools
- need: stop of "leaks" of exploitation tools

Goals of binary encryption

- allow execution only for authorized persons
- obscure purpose of the binary
- immune to static analysis (IDA, objdump, ..)
- obscure process image (SIGSTOP, /proc/, core)

Forensics - status quo

Almost all forensic tools on UNIX are

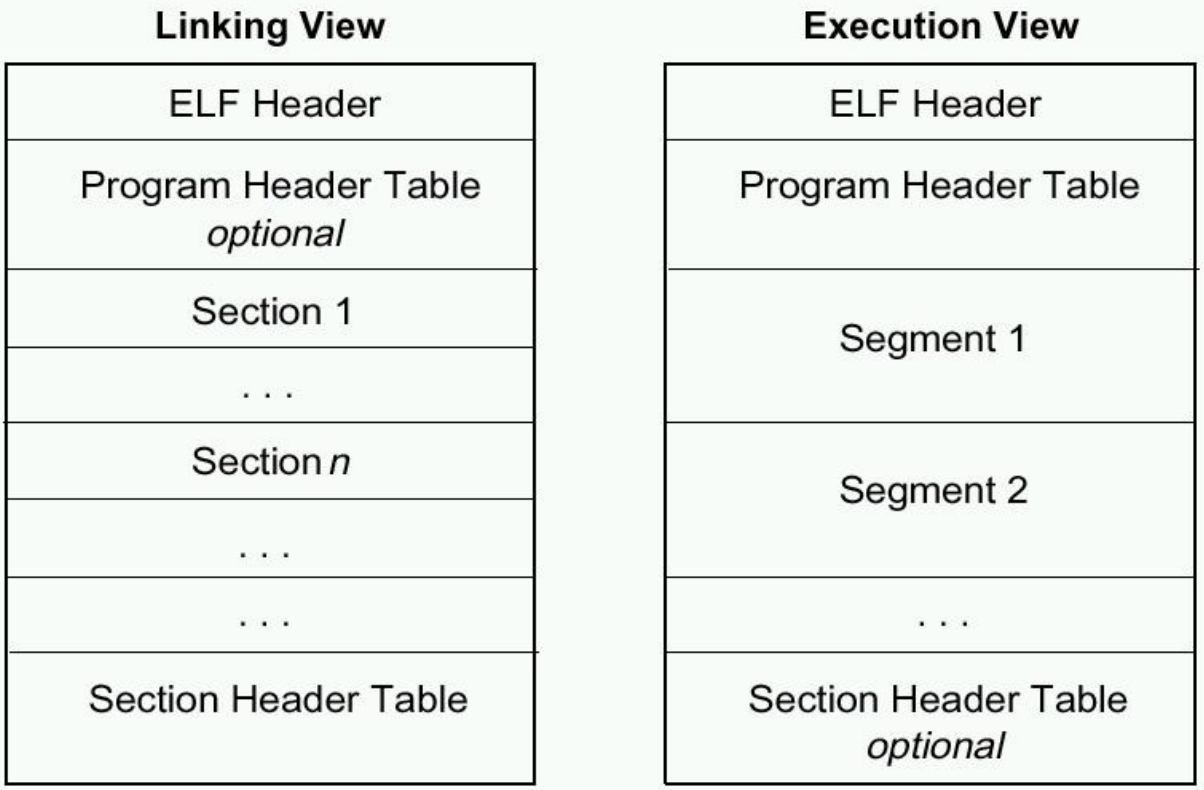
- intended for debugging (gdb, strace, strings, objdump, ..)
- fail to deal with hostile code (libbfd/ptrace based programs)
- obscure and not well documented
- old and buggy (TCT, ptrace-interface, ..)

See http://www.incidents.org/papers/ssh_exploit.pdf to see the failure of such tools.

ELF file format

- standard UNIX executeable format (TIS ELF v1.2)
- used for both linking objects and executeables
- standard-based, well designed
- used in: Linux, FreeBSD, IRIX, Solaris

ELF dualism



OSD1980

Example ELF file

- "readelf -l /bin/ls"
- two PT_LOAD segments (code, data)
- one PT_INTERP segment ("/lib/ld-linux.so.2")
- entry point from ELF header

ELF program header

```
typedef struct
{
  Elf32_Word    p_type;         /* Segment type */
  Elf32_Off     p_offset;       /* Segment file offset */
  Elf32_Addr    p_vaddr;        /* Segment virtual address */
  Elf32_Addr    p_paddr;        /* Segment physical address */
  Elf32_Word    p_filesz;       /* Segment size in file */
  Elf32_Word    p_memsz;        /* Segment size in memory */
  Elf32_Word    p_flags;        /* Segment flags */
  Elf32_Word    p_align;        /* Segment alignment */
} Elf32_Phdr;
```

- p_type: PT_LOAD, PT_INTERP, PT_NOTE, PT_DYNAMIC, ...
- p_vaddr: real absolute memory start address
- p_flags: PF_R, PF_W, PF_X
- details: /usr/include/elf.h

ELF - the linking view

- file type ET_DYN
- libraries are relocateable object files
- relocation information in file
- additional symbol information stored
- dynamic section required

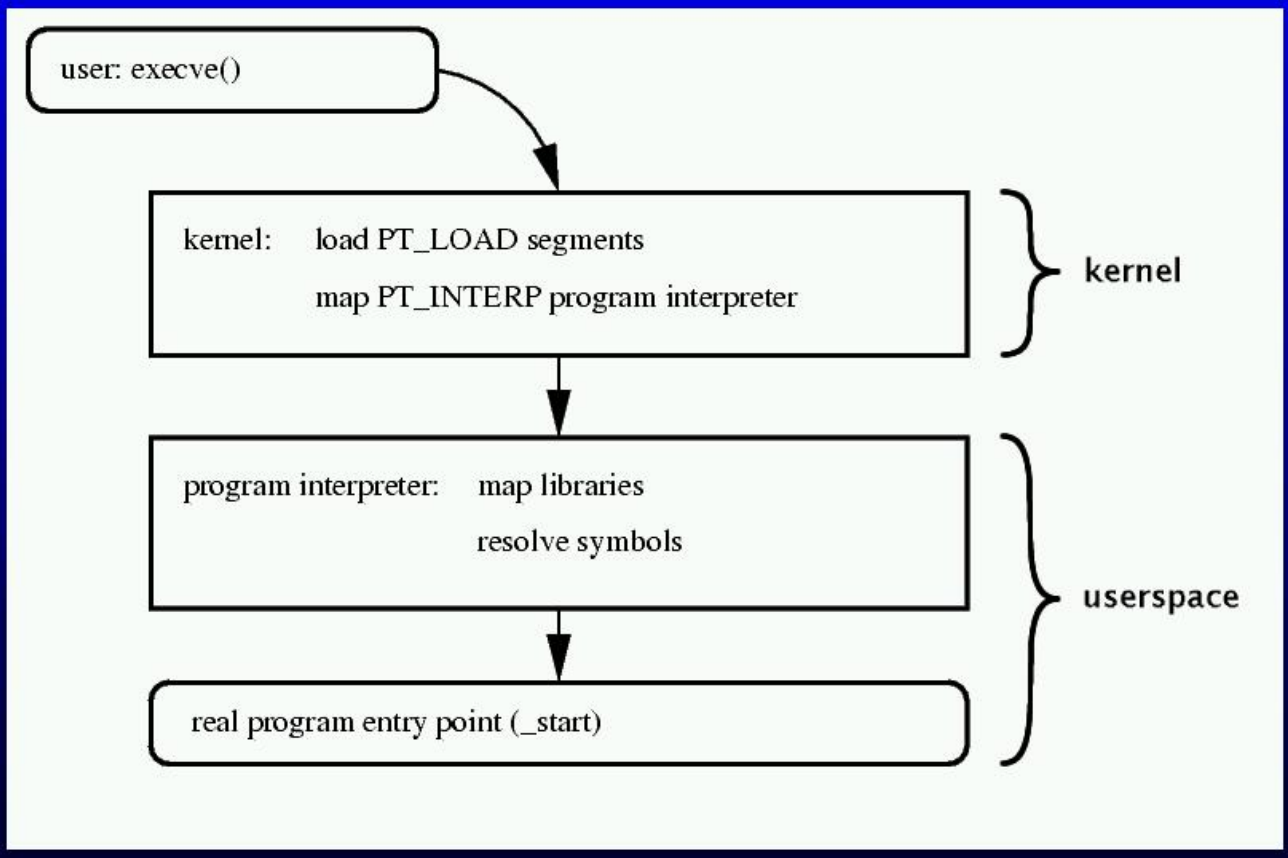
ELF loading

- execve() call executes an ELF
- teamwork: kernel and userspace

Distribution of work:

- kernelspace: mapping executeable and program interpreter
- userspace: mapping libraries, resolving dependancies

ELF loading (Linux)



Kernel ELF loading

- look through all segments
- map PT_LOAD segments into memory
- map program interpreter from PT_INTERP segment
- control to userspace: start program interpreter

Userspace ELF loading

Program interpreter (PT_INTERP)

- receives control from kernel
- parameters in Elf32_auxv vectors
- loads all libraries, resolves all symbols
- pass control to real entry point (ELF header)

additional code: typical ELF virii

- ELF PT_LOAD segments are page-aligned
- segment padding is needed
- add code into padding
- redirect entry point

More sophisticated ways do exist, see silvio's papers.

additional code: userspace ELF loader

- used first by UPX packer
- minimal ELF stub
- stub works as kernel-alike ELF loader
- pros: little overhead, reliable
- cons: slowdown, weak protection

The future

- forensic work will become more difficult
- todays forensics will drop out
- reverse engineers will convert to UNIX
- combination: binary encryption, worms, virii
- development: tougher analysis tools
- development: stronger protections

The end

Documentation

- http://www.team-teso.net/articles/18c3-encryption/

Contact

- scut@team-teso.net

Thank you for your interest :-)