Shiva
Advances in ELF Binary Encryption

Shiva Authors: Shaun Clowes, Neel Mehta

Presented by:
Neel Mehta
(nmehta@iss.net)

Runtime Binary Encryption

- An executable executes as normal, but is encrypted on disk.
- Resistant to analysis and modification.

ELF

- Executable and Linkable Format
- The executable file format on virtually all modern UNIX platforms.
  - Header
  - Sections / segments.
  - Symbols, string tables, relocations.

Runtime Binary Encryption: A VERY Brief History

- Mainly confined to MS platforms.
- 1980’s, early 90’s - .COM
  - 0x100

Windows

- Pe-Crypt – 1998 – random, acpizer, killa
- Now dominated by ASPack, UPX, other commercial encryptors.
- Very commonly used in malware of all kinds.

Unix

- Burneye – 2001 – Scut
- UPX now supports Linux.

Shiva - 2003

- Shaun Clowes and Neel Mehta
- Designed to bring many of the advanced techniques from Windows to Unix, as well as many new techniques not implemented elsewhere.
- Designed to encrypt Linux/x86 executables.

The Encryptor's Dilemma:

To be able to execute, a program's code must eventually be decrypted

Binary Encryption: An Arms Race

- Thus binary encryption is fundamentally a race between developers and reverse engineers.
- The encryptors cannot win in the end
  - Just make life hard for the determined and skilled attacker.
  - Novices will be discouraged and look elsewhere.

Encryption Keys

- If the encrypted executable has access to the encryption keys for the image:
  - By definition a solid attack must be able to retrieve those keys and decrypt the program
- To reiterate, binary encryption can only *slow* a determined attacker

Our Aim

- Introduce some novel new techniques.
- Advance the state of the art:
  - Unix executable encryption technology trails Windows dramatically
- Promote interest in Reverse Engineering on Unix platforms

What’s the point?

- An encryptor can be used to:
  - Prevent trivial reverse engineering of algorithms
  - Protect setuid programs (with passwords)
  - Hide sensitive data/code in programs

Standard Attacks

- A good encryptor will try to deter standard attacks:
  - strace – System Call Tracing
  - ltrace – Library Call Tracing
  - fenris – Execution Path Tracing
  - gdb – Application Level Debugging
  - /proc – Memory Dumping
  - strings – Don’t Ask

Deterring Standard Attacks

- strings
  - Encrypting the binary image in any manner will scramble the strings

Deterring Standard Attacks

- ltrace, strace, fenris and gdb
  - These tools are all based around the ptrace() debugging API
  - Making that API ineffective against encrypted binaries would be a big step towards making them difficult to attack

Deterring Standard Attacks

- /proc memory dumping
  - Based on the idea that the memory image of the running process must contain the unencrypted executable
  - A logical fallacy
  - A good encryptor will invalidate it

Countermeasures

- The majority of attacks against encrypted executables (excluding static analysis) can be detected by the running program
- Unless the attacker notices and prevents it, the program can take offensive action

A Layered Approach.

- Static analysis is significantly harder if the executable is encrypted on more than one level
- The layers act like an onion skin
- The attacker must strip each layer of the onion before beginning work on the next level

(Un) Predictable Behavior

- Efforts to make encryptor behavior differ from one executable to another are worthwhile
- The less generic the methodology, the harder it is to create a generic unwrapper

Shiva's Features

- The encyptor we'll present today tries to implement all of the defences we've described so far.

Shiva v0.99

- Currently encrypts dynamic or static Linux ELF executables
- Does not handle shared libraries (yet)

Encryptor / Decryptor

- Development of an ELF encryptor is really two separate programs
- Symmetrical operation

Encryptor

- Normal executable, which performs the encryption process, wrapping the target executable

Decryptor

- Statically-linked executable, which performs decryption and handles runtime processing
- Embedded within the encrypted executable
- Self contained
  - Cannot link with libc etc.

Shiva ELF Abstraction API

- Represent any ELF executable as a structure in memory
- Allows for easy manipulation of ELF executables within encryptor, not relevant for decryptor

Dual-process Model (Evil Clone)

- Slave process (main executable thread) creates a controller process (the clone)
- Inter-ptrace (functional and anti-debug)

x86 Assembly Byte-Code Generation

- Allows for the generation of x86 assembly byte-code from within C (a basic assembler)
- Pseudo-random code generation, pseudo-random functionality

Encryption Layers – Layer 1

Obfuscation Layer
Obfuscated

Initial Obfuscation Layer

- Intended to be simple, to evade simple static analysis
- Somewhat random, generated completely by in-line ASM byte-code generation

Encryption Layers – Layer 2

Obfuscation Layer
Password Layer
AES Encrypted

Password Layer

- Optional
- Wrap entire executable with 128-bit AES encryption
- Key is SHA1 password hash, only as strong as the password

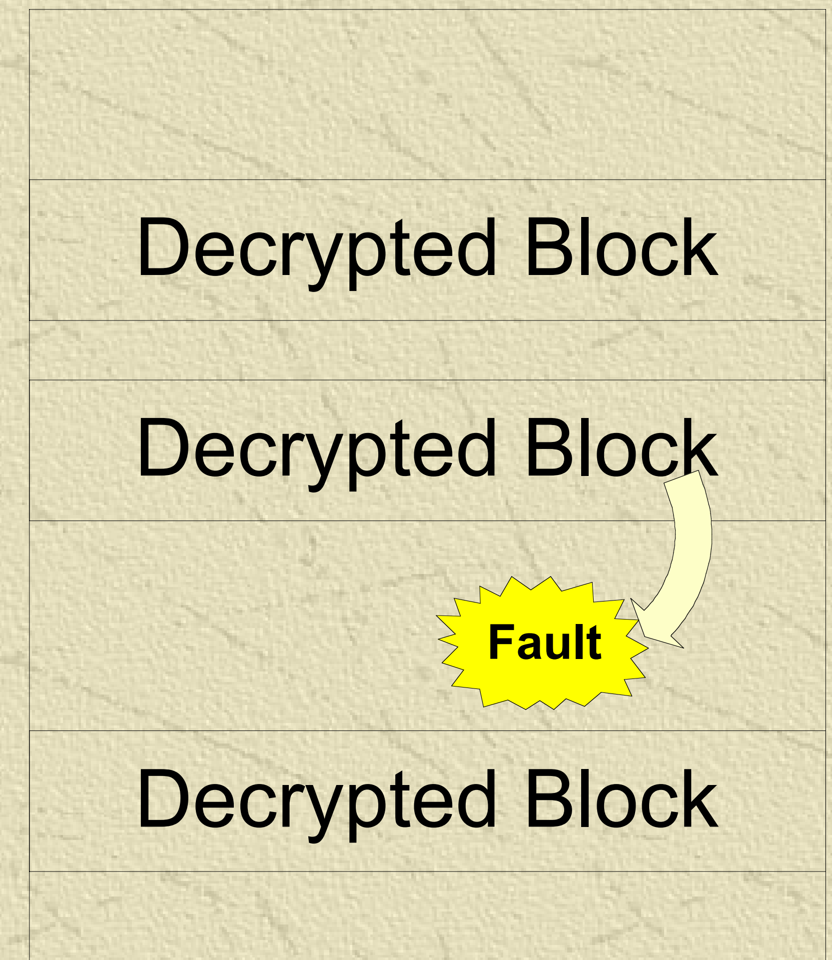
Encryption Layers – Layer 3

Obfuscation Layer
Password Layer
Crypt Block Layer
Crypt Blocks

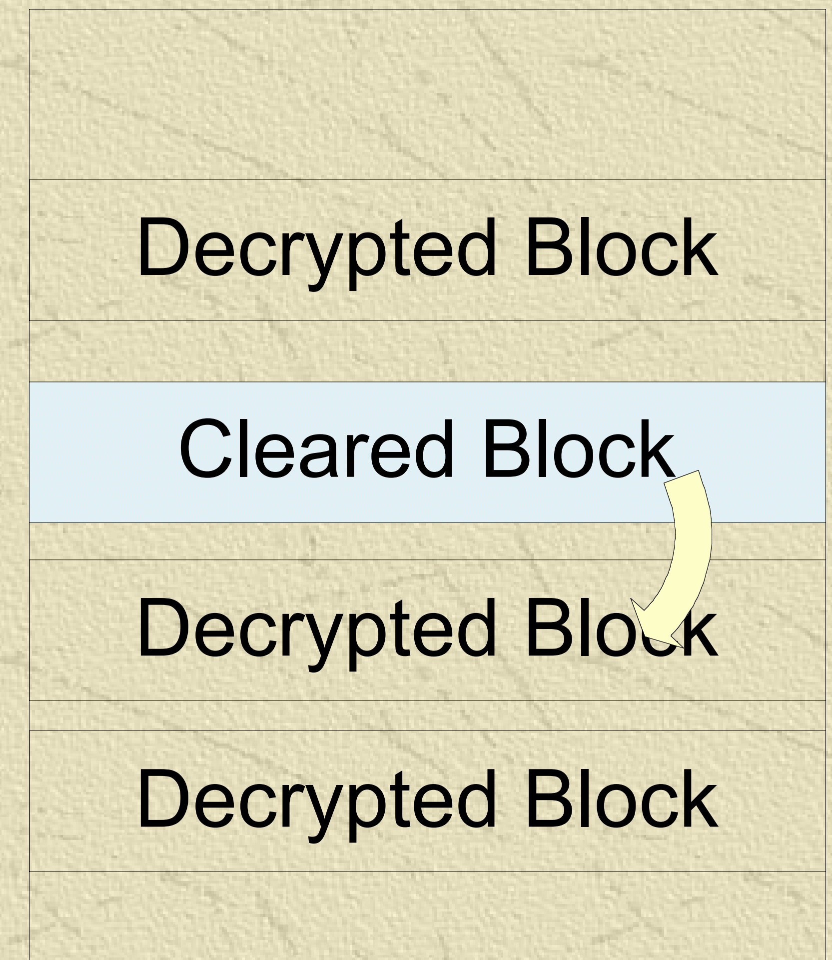
Crypt Blocks

- Two important types – immediate map, map on-demand
- Controller process handles map on-demand blocks
- Random unmap
  - Only small portion of executable decrypted at any time
- Instruction length parsing – necessary to create map on-demand blocks

Crypt Block Mapping



Crypt Block Mapping



Crypt Block Encryption

- Block content encrypted with strong algorithm
  - Guess
- Code to generate keys made pseudo-randomly on the fly (asm byte-code)
  - Keys are never stored in plain text
- Tries to bind itself to a specific location in memory (and other memory context)

Dynamically Linked ELF’s

- Decryptor interacts with system’s dynamic linker
- Decryptor must map dynamic linker itself, and then regain control after linker is done

Anti-debugging/disassembly

- Inherent anti-debugging provided by dual-ptrace – link verified
- Catch tracing:
  - Check eflags
  - Check /proc/self/stat

Anti-debugging/disassembly

- Timing and SIGTRAP
- Simple SIGTRAP catch
- JMP into instructions – common anti-disassembly trick

Byte-Code Manipulation: Beyond ELF

- Currently x86 specific.
- Requires significant code analysis.
  - Instruction by instruction processing.
  - Function recognition, code flow analysis.
  - Requires a fairly well designed and implemented framework.

Easy Ways to Manipulate Byte Code

- Call redirection.
- Jump redirection.
- Jmp tables.
- Other constructs.

Instruction Emulation

- Easily accomplished via manipulating ptrace register structures.
- Virtually every instruction can be emulated if its operation is understood.

Problems Encountered, Solutions

- Clone, ptrace, and signals
- Fork processing
- Exec processing
- Life without libc
  - Simple implementations of malloc etc

Current Limitations

- Can’t handle vfork(), threads
- Can’t encrypt static executables that call fork()
- On Linux, exec() fails if the calling process tries to exec a setuid program
- Section Headers
- Nothing that can’t be solved by the next release ☺

Shiva in Action

Demo

Future Direction

- Ports to other OS's/Architectures
- Support for shared libraries
- Advanced anti-debugging
- Adapting when people break it

End of Presentation

- Thanks for listening
- Questions?