Fixing/Making Holes in Binaries

• The Easy, The Hard, The Time Consuming

Shaun Clowes – shaun@securereality.com.au

What are we doing?

Changing the behaviour of programs
 Directly modifying the program in it's compiled (binary) form
 No source code
 No recompilation

Types of Binary Modification

- Two types:
 - Static (executable file) modification
 - Runtime (process) modification

Why modify binaries?

For the usual reasons for working on software: Fix bugs Add functionality Change behaviour

Why not just use source?

No access to it
 COTS
 Not readily available
 Source code on a disk hidden in a cupboard behind mountains of other disks and CDs
 Running programs

99.999% uptime, no restart

Isn't it too hard?

Traditionally been in the too hard basket Depends on the objective Normal value proposition It can be easy, it can also be very hard We'll cover a variety of techniques All do need some coding skill

How's this related to Security?

Two aspects
 Defender

 Fix holes in vulnerable software
 No waiting for vendor patches

 Attacker

 Backdoor/Trojan software



- Unix systems
 - Solaris
 - Linux
- ELF binaries
- Most concepts more generally applicable

Where to from here?

All about ELF
File patching
In memory patching
Library interception
injectso - Run time library interception

Breakdown of ELF

 Need understanding of internal structure of executables
 ELF = Executable and Linkable Format
 Originally by Unix System Labs (USL)
 Adopted by Tool Interface Standards committee (TIS)

Used in virtually every recent Unix

Breakdown of ELF

 Three main types of ELF files
 Relocatable file – object file ready to be linked with others
 Executable
 Shared Object (Library)
 Only last two relevant
 Concentrate on Executables

Static Vs Dynamic Executables

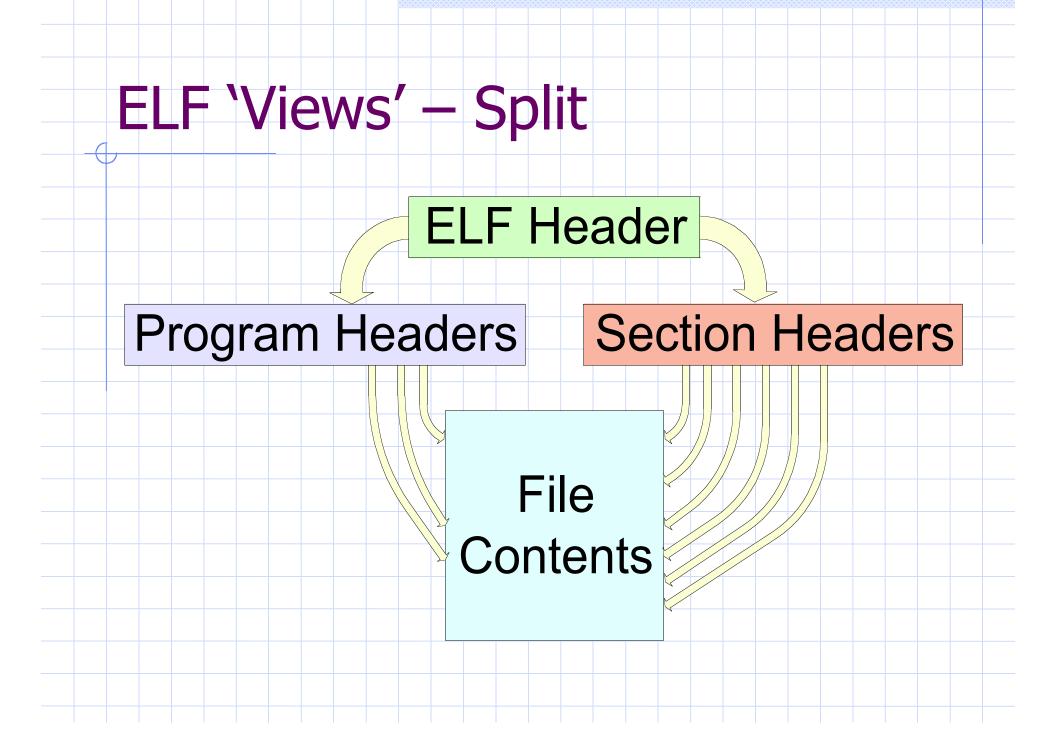
Static executables are self contained They do not need any external code Dynamic executables use external data/code at run time Shared libraries Smaller executables Less disk/memory use Much more common

ELF 'Views'

ELF describes two separate 'views' of an executable, a linking view and a loading view

Linking view is used at static link time to combine relocatable files

Loading view is used at run time to load and execute program



ELF Linking View

 \bigcirc

.interp	Divides executable into
.dynamic	many meaningful 'Sections'
.dynsym	Sections have:
.dynstr	A name and type
.rel.plt	 Requested memory location at run time
.text	 Permissions
	(writable/executable)

ELF Linking View – Important Sections

 .interp Requested Dynamic linker Dynamic linking information .dynamic Symbols (static/dynamic) .symtab, •dytreater, .dynstr String tables Procedure linkage table •.plt •.rel.<x> Relocations for section x Code •.text Initialized data .data

ELF Linking View

Not all sections needed at run time
 Information used for linking
 Debugging information
 Difference between link time and run time

ELF Loading View

1

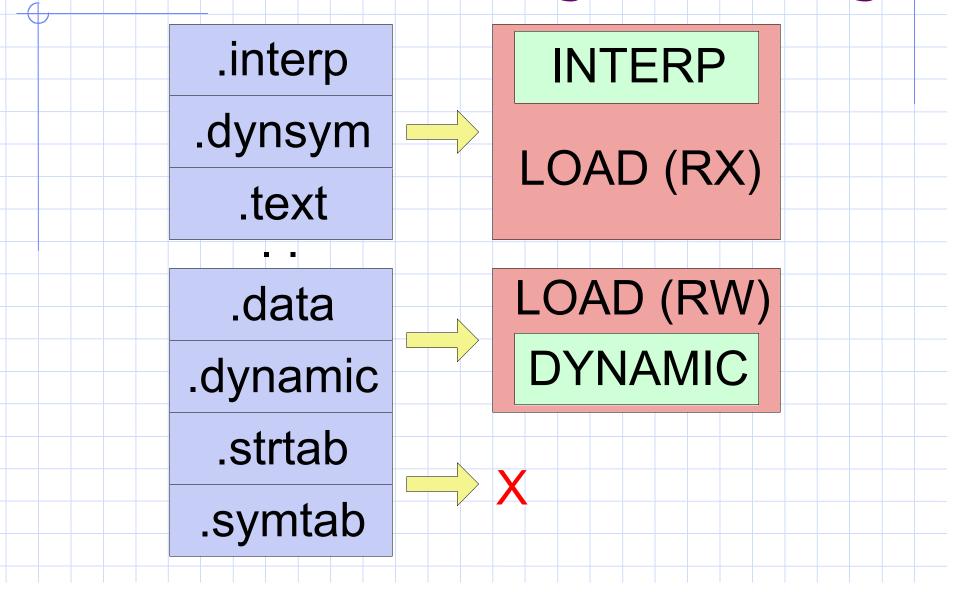
	Much simpler view, divides
INTERP	executable into 'Segments'
LOAD	Describes
	 Parts of file to be loaded into memory at run time
LOAD	 Locations of important data at run
DYNAMIC	time
	Segments have:
	A simple type
	Requested memory location
	Permissions (R/W/X)
	 Size (in file and in memory)

ELF Loading View – Segment

Types

- •LOAD •Portion of file to be loaded into memory
- •INTERP •Pointer to dynamic linker for this executable (.interp section)
- •DYNAMI •Pointer to dynamic linking C information (.dynamic section)

ELF 'Views' - Linking to Loading



ELF Loading View

Semantics of section table (Linking View) are irrelevant in Loading View

- Section information can be removed from executable
 - Good way to kill GNU libbfd programs

ELF Loaders

 Operating system routines to load executable and begin execution
 ELF is very flexible, Loaders aren't
 Bugs and idiosyncrasies
 ELF files conforming to the specification don't always run

Loading and Executing an ELF Executable

- File opened
- Map LOAD segments into to memory
 - Calls the dynamic linker specified in the INTERP segment, passing information about the executable

Dynamic Linker/Loader

Handles all of the dynamic/shared library needs of executable

- Retrieves information from the DYNAMIC segment
- Loads all required shared libraries into memory

Modifies executable such that it can access needed resources in the libraries

The Dynamic Section/Segment

A table with records containing data critical to dynamic loading/linking Allows dynamic linker to quickly find out information about the executable No need for section table etc Each record consists of: A type (tag) Value (or pointer)

Dynamic Segment Record

Tags

•DT_NEEDED •Offset to name of a required shared library

- •DT_REL •Address of relocation entries
- •DT_JMPREL •Address of relocation entries associated with the PLT

•DT_DEBUG •Pointer to debugging information from dynamic linker

Loading and Executing an ELF Executable

Map in shared libraries corresponding to DT_NEEDED entries

Add libraries to link map stored in area referenced by DT_DEBUG entry

Perform relocations

Relocations

 Tell Dynamic Linker to rewrite parts of executable to refer to external resources

 Link to dynamic symbol table entries

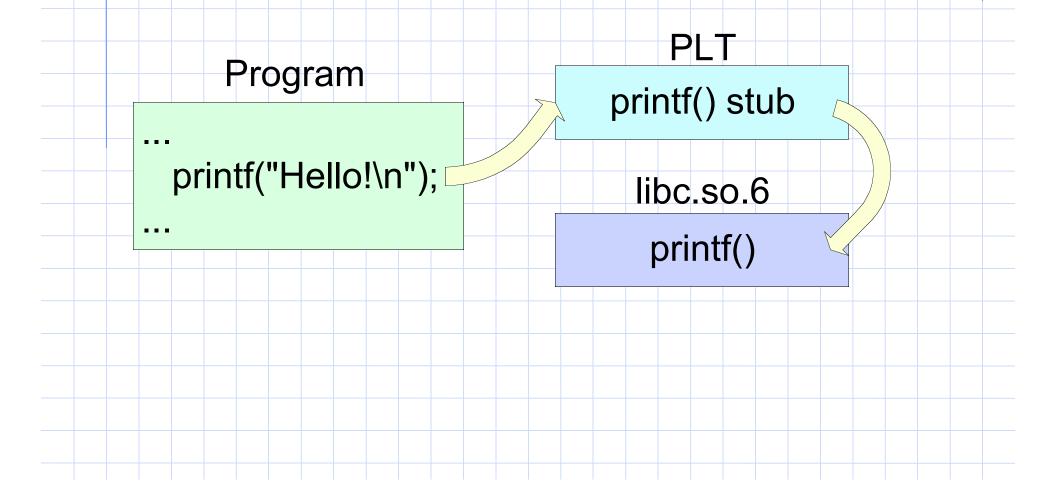
 Needed to allow program to use code/data in shared libraries

 Since address decided at run time

The Procedure Linkage Table

Stored in the .plt section Allows executables to call functions that aren't present at compile time Shared library functions (e.g printf()) Set of function stubs Relocations point them to real location of the functions Normally relocated `lazily'

The Procedure Linkage Table



The Global Offset Table

Like PLT but for non function symbols

 `stderr', `errno' etc

 Referenced by PLT on IA32

 But NOT Sparc

 Both PLT and GOT targeted for attack in format string vulnerabilities

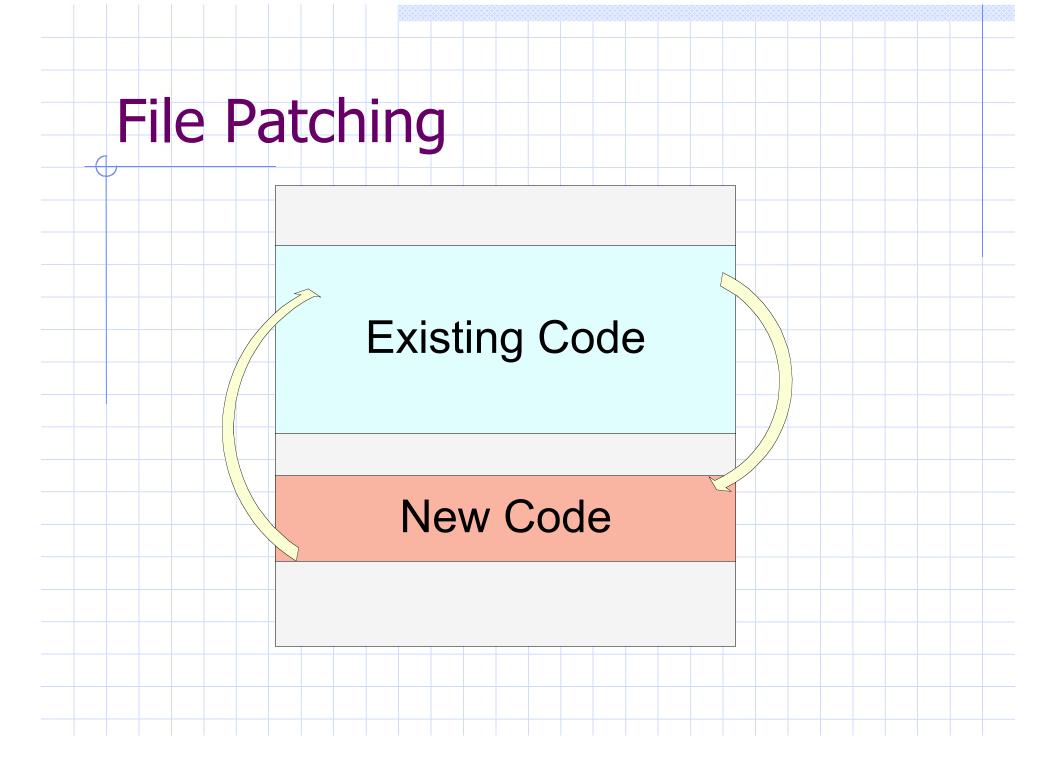
DT_DEBUG Record

 Pointer to a debug structure provided by the Dynamic Linker (at run time)
 Normally examined by Debuggers
 Contains pointer to link map which describes memory layout of process

 Which binary files are loaded
 Base address at which they are loaded

File Patching

- Statically modify code in executable file
- Need to:
 - Insert additional code
 - Link existing code to added code



File Patching

Need to understand existing code
 Disassembly
 Reverse Engineering
 New code
 Assembly
 Hybrid C

File Patching

Where to put additional code?
 Overwrite existing unused code
 Hard to identify
 Section padding
 Not much space
 Need section in executable segment (on non IA32 targets)

File Patching

Add a segment

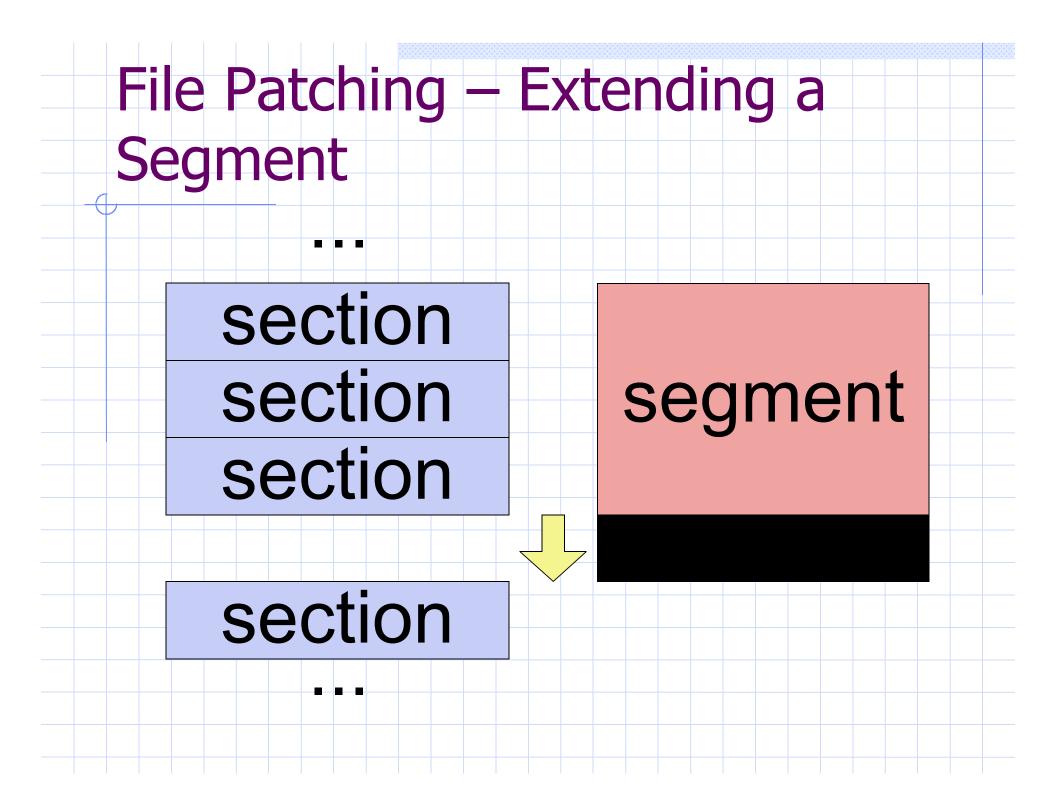
 ELF Loader bugs and issues

 Extend an existing segment

 Add as much space as you need

 Other methods

 Unix viruses



File Patching



File Patching

- Features
 - Very powerful, can change almost anything
 - Permanent
- But.
 - Complex and error prone
 - Program must be restarted
 - Can't easily call new library functions

In Core Patching

Exactly like file patching but performed on process memory image

Modify process memory using ptrace() or procfs

In Core Patching

Where to put additional code?
 Memory space must be mapped executable

 Except under IA32
 Overwrite unused code
 Hard to identify
 Use segment padding

 Segments padded to page boundaries

In Core Patching – Segment Alignment

•File Size	 Memory Size
•0x000f0	•0x00110
padding	1604 Bytes
segment	240 Bytes
nulls	32 Bytes
padding	2220 Bytes
	 •0x000f0 padding segment nulls



In Core Patching

- Features
 - Very powerful, can change almost anything
 - Non permanent
 - Can be performed on running process
- ♦But.
 - Complex and error prone
 - Can easily kill target
 - Limited space for new code
 - Can't easily call new library functions

Library Interception

 Dynamic loader resolves at run time all external symbols (dynamic relocations)
 GOT – Data relocations
 PLT – Function relocations

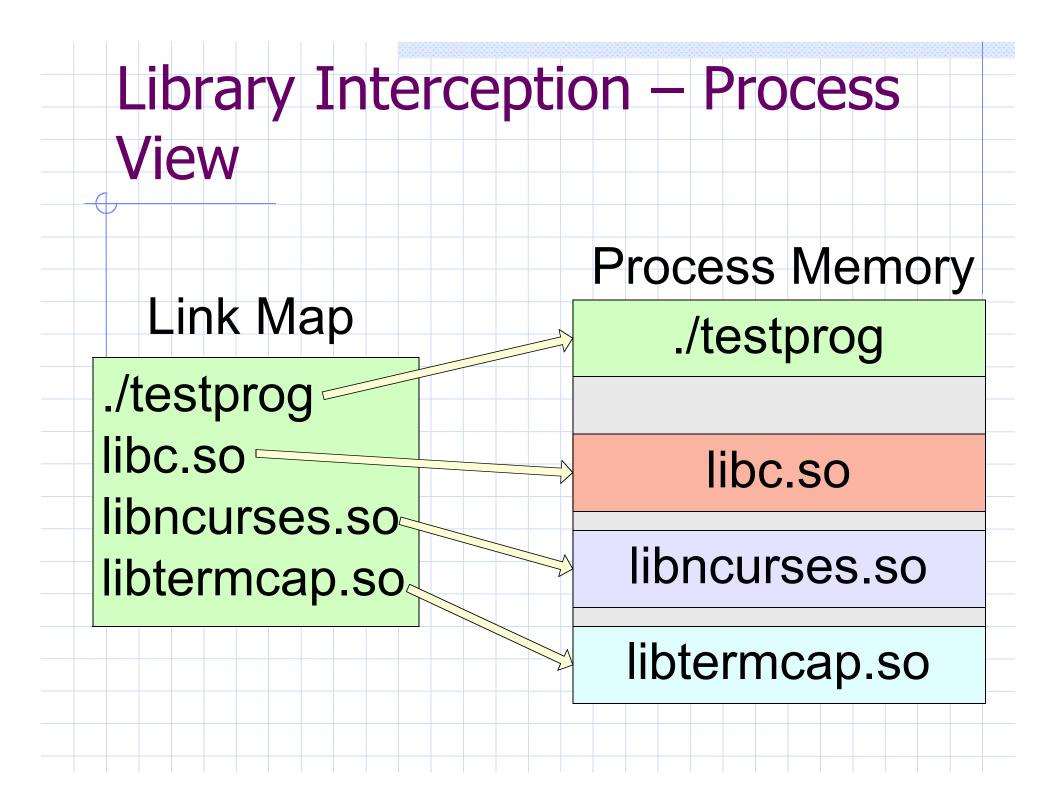
How?

Library Interception

Reads DT_NEEDED entries in PT_DYNAMIC segment

Loads files and adds them to link map

Then goes on to process relocations



Dynamic Linker - Resolution

 When processing relocations dynamic linker looks through map and
 Searches each libraries exported symbols
 Tries to find matching symbol name
 Looks for non 'weak' function
 First match is the winner Library Function Call Interception

Trick is to get your library loaded first

It will be the match (winner) for all its exported symbols

 Can intercept any dynamic function call (libc etc)

Library Interception – Getting in First

Modify DT_NEEDED records

Overwrite other library entry

- Open it in your library with linker routines
- Substitute library depends on old library
- Move DYNAMIC segment and recreate
 - Add entirely new library dependency

Library Interception – Getting in First

- Use Linker environment
 - LD_PRELOAD specifies libraries to be loaded immediately
 - Very common technique

Library Intercpetion – Calling Real Function

- Intercepting function usually needs to call old function
- Dynamic linker provides interface (through libdl.so):
 - dlopen Open a library
 - dlsym Get address of symbol,
 RTLD_NEXT flag specifies libraries after
 current one

Library Interception - Demo



Library Interception



- Easy and simple
 - All interception code can be done in C
- Safe
- Can call any library functions easily

Library Interception



- LD_PRELOAD not permanent
- DT_NEEDED approach library in place at all times
- Program must be restarted

injectso – Runtime Library Interception

injectso is like InjLib for Windows
 Injects a shared library into a running program

Much harder on Unix than in Windows

Operating system provides minimal support

injectso - Breakdown

- Opens process (using ptrace() or procfs)
- Searches link map to find dlopen() equivalent
- Construct arguments to dlopen() on stack
- Force process to jump to dlopen()
 - Set return address to be 0x41414140

injectso – Breakdown

Function ends with SEGFAULT

 Return to 0x41414140
 Intercepted by injectso

 Can call user function

 Provide address of DYNAMIC segment

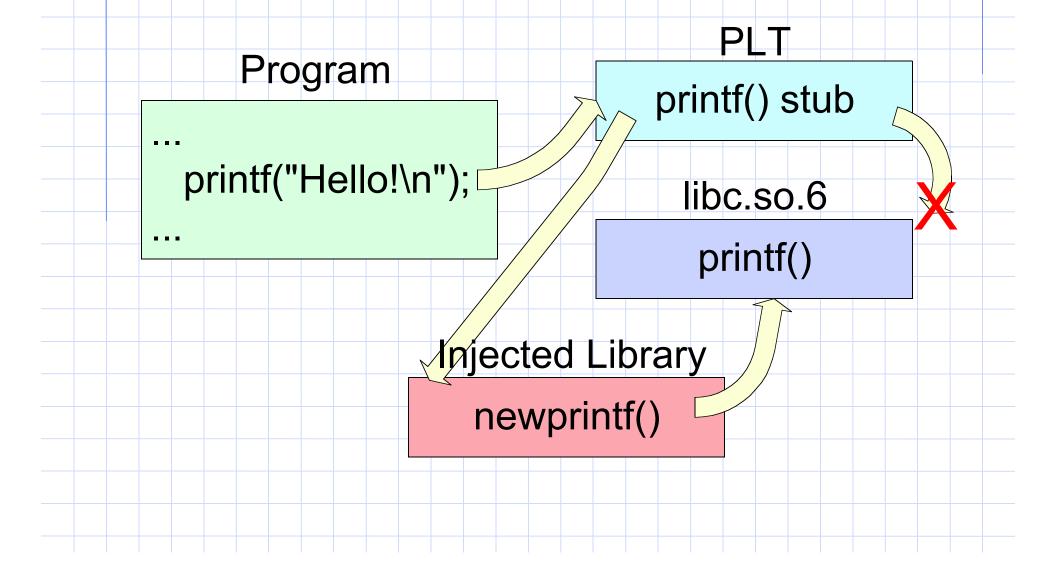
 Process state restored

 Syscalls restarted

injectso – Intercept.o

Utility object, link into inject libraries Redirect dynamic function calls User code provides list of functions to override Finds functions in PLT and patches them to refer to replacement versions Provides pointer to old function











- Simple
- Flexible
- No modifications to binary file
- No disk files
 - Library can be deleted after injection
- Service does not need to be restarted

injectso



- 'Unlimited' space for code
- No trampolines
 - Executable/Writable memory not required
- Can call any library functions easily
- Can override any library functions easily



Not permanent

Thankyou for listening!



- Feedback
 - IP 172.16.0.2
 - MAC 00:10:5a:d3:5d:b2
 - SSL Certificate Fingerprint (SHA1)
 - a0 8f db 28 7a 15 2e 86 76 42
 - 6e 8a b1 27 32 55 4f 31 12 06
 - Username and Password are printed on the receipt part of your badge

Contact Details



http://www.securereality.com.au



shaun@securereality.com.au