Generic Technical Defences

Shaun Clowes
SecureReality
Introduction

- Techniques for mitigating threats
  - Even unknown ones
- Host Based
- Generic
Overview

- Relating host security to the human body
- Immune System – Generic Protections
- Biodiversity – Being Different
Relating a Host to a Human Body

- A single vulnerability is like one specific illness
  - A patch is like a medical cure for that one illness
- There are many unknown illnesses and previously unseen variants of known illnesses
  - The immune system tries to adapt
- Even an illness the immune system cannot stop requires conditions to be right
  - Natural diversity can stop them in their tracks
Enterprise Security

- Most enterprises manage security with:
  - Firewalls
  - Network IDS systems
  - Patching
Firewalls/IDS

- Firewalls and Network IDS systems are at the perimeter
- Not helpful for new attacks over vectors considered safe
- In our body analogy we could say that they act like covering your mouth when somebody else sneezes
Patching

- Patching is rapidly becoming an impossible task
  - Particularly since they need to be well tested to insure they don’t break other things
- Once a “virus” makes it past the IDS and into an unpatched vulnerable application it’s open season
  - A host has a soft underbelly
Attacks as “Viruses”

- An attack making it past the firewall and IDS is like Flu particles making it into a body's system.
- The body will develop an infection unless the immune system stops it or the host is otherwise unsuitable.
Technical Protections

- Software to harden the OS/Application
- Effectively add an "immune" system to a host
- Don’t need to be deployed alone
  - Deployed in concert they can have an effect greater than the sum of the parts
Immune System Failures

- Even a strong immune system will not resist every attack
  - Particularly not attacks of a kind never seen before

- Technical protections are the same
  - They serve to make it much more difficult for attacks
  - Particularly for known attack vectors
Soft Targets

- Illness has a tendency to afflict the frail, and to do so with greater severity, than it does the healthy.
- Computer security is much the same.
  - The majority of computer crime, like crime in real life, is crime of opportunity.
- A host that has a strong immune system becomes a far less interesting target.
Making Attacks Ineffective

● Most technical protections aim to stop one or more key aspects of known attack classes
● We’ll first quickly cover some attack classes
  - How they work, key attributes for attack success
● We’ll then discuss a number of generic protections:
  - How they work, how they can be bypassed
  - All can be applied to system without source code
Stack Overflow

- Smashing the Stack for Fun and Profit
- Important data on the stack is overwritten
  - Resulting state can be controlled in some way by the attacker
- Traditional example is a string copy of a string from a user into a stack buffer
- Not particularly common any more
Traditional Stack Smash

Function A()

Function B()

B(UserString)

A’s Local Variables

Saved Base

Return Addr

*UserString

A’s Local Variables

Base Pointer

Base Pointer

Stack

Stack
Traditional Stack Smash

Function B()

char buffer[100];
strcpy(buffer, UserString);
Traditional Stack Smash

Function B()

Stack

Char buffer[100];
strcpy(buffer, UserString);
Traditional Stack Smash

- At this point the attacker can redirect execution as they wish
  - Address usually cannot contain nulls
  - Other restrictions may be enforced (e.g. input filters)
- Usually want to execute arbitrary code
- Redirect execution to an input buffer
  - Fill that input buffer with executable machine code
Traditional Stack Smash

Function B()

UserString

>= 108 Bytes

= Code

Overwritten

Saved Base

Overwritten Return Addr
Traditional Stack Smash

- The attacker’s code does **not** have to be stored in the overflowed stack buffer
  - It just needs to be at a reasonably predictable address
  - Must be in an executable memory page
  - Stack and Static buffers are the most convenient for this
    - Static buffers are at an *absolute* address
  - Some heap buffers are also highly predictable
Format String Attack

- Simple programming error
- Becoming less and less common
- Functions with variable number of arguments cannot know how many they were passed
  - Have to be given guidance
  - If the guidance is wrong (i.e., indicates more arguments than there are) function will just keep walking down the stack operating on unrelated data
Format String Attack

- Formatted output functions (fprintf, printf, sprintf etc) use a “format” string argument
  - Format string is directly copied to the output except for “%<format><type>” tokens which are substituted with a string created from the next argument
  - One format type *writes* to an address specified in an argument (%n), it writes the number of bytes that would have been copied to the output so far
Format String Attack

- If a programmer wants to output or otherwise copy a user string they may accidentally specify it as the format argument
  - This will work perfectly as long as there are no % modifiers in it
  - printf(UserString); vs printf(“%s”, UserString);
Format String Attack

- If the attacker can control the format string they can use %n to write arbitrary data to arbitrary addresses
  - Can’t normally have nulls in data/addresses
  - Need to have known control over some stack values in order to be able to specify the target address
Format String Attack

- Redirect execution to hostile code by overwriting:
  - Return address on stack (jump into one of their buffers)
  - A GOT/PLT entry (used to call external functions, known location)
  - Function pointer somewhere in program
  - atexit pointers
  - … the list is endless
PLT/GOT Overwrite

Program Image

Code

... printf("Hello!\n");

libc Image

... printf();

PLT

printf() Stub
PLT/GOT Overwrite

Program Image

Code

... printf("Hello!\n"); ...

PLT

printf() Stub

Stack

printf()

User Buffer
Heap Overwrites

- The system allocates memory to processes in large blocks (page sized)
- But applications typically want to allocate small blocks of memory for miscellaneous storage
- A heap implementation manages dividing the large blocks of memory for use
  - Which parts are in use and which aren’t
Heap Overwrites

- The information about the blocks must be stored somewhere
  - Makes most sense to store it in the bytes immediately around the block itself
- The MetaData is usually used not just to describe the block nearby but also to link the blocks together
Heap Overwrites

- If the meta data can be overwritten the logic that links the blocks together can often be manipulated to overwrite arbitrary memory locations
  - Fooling the heap implementation into thinking the target area is a block
- Similar situation as for format strings results
Integer Evaluation Problems

- Getting a lot of attention lately
- Based around C/C++ integer calculation characteristics
  - \( int \ 2147483647 + 1 = -2147483648 \)
  - \( unsigned \ int \ 4294967295 + 1 = 0 \)
  - \( int \ 65536 = short \ 0 \)
  - \( int \ 1073741825 \times \text{sizeof}(int) = 4 \)
Integer Evaluation Problems

- Two common scenarios:
- Fool program into allocating a much smaller block of memory than needed, resulting in a heap overwrite
  - Usually involves providing a large number which later calculations cause to loop back to a small number
  - Opens door for heap overwrites
Integer Evaluation Problems

- Fool program into writing to memory outside the bounds of a block
  - Usually when the integer is used as an array index, providing a negative number that passes $x < \text{array length}$ tests
  - Often results in ability to write data to a selection of memory address
Most Common Factors

- A user buffer at a reasonably predictable location
  - In memory with execute permission
- Use of vulnerable standard functions
  - strcpy(), strncpy() etc
- The ability to access standard system services in hostile code
  - fork(), exec(), open() etc
Libsafe

http://www.research.avayalabs.com/project/libsafe/

- Shared library that intercepts calls to vulnerable functions
  - strcpy(), strncpy(), stpcpy(), wcscpy(), wcpcpy(), strcat(), strncat(), wcscat(), vfprintf(), sprintf(), vsprintf(), vsnprintf(), gets(), getwd(), realpath(), fscanf(), vfscanf(), sscanf()

- Loaded via LD_PRELOAD environment variable or /etc/ld.so.preload file
  - Loaded before libc, application transparently uses the replacements
Libsafe

- Replacement functions check if any of the destination locations are on the stack
  - If they are verifies they’re **within** a frame (i.e saved frame pointers excluded)
  - If the destination would overwrite a frame a stack trace is generated, a log is sent to syslog() and the program is terminated with SIGKILL

- Prevents stack frame smashing via the vulnerable functions
Function B()

```c
char buffer[100];
strcpy(buffer, UserString);
```
Libsafe - Strengths

- Very easy to install and enable
- Minimal impact on applications/system
Libsafe - Weaknesses

- Protects *only* the specified functions against *only* stack frame overwrites
  - The vulnerable functions may still overwrite other important data structures (e.g. heap overflows, format string exploits)
  - Does not protect any application functions
- Cannot protect code compiled with -fomit-frame-pointers
Libsafe - Weaknesses

- Format string parsing code is buggy, “%1n” can bypass checks
- While libsafe is limited in scope, its adverse impact is so minimal that deploying it is a nil sum equation
Openwall Kernel Patches

http://www.openwall.com/linux/

- Part of the Openwall Linux distribution
- Set of patches to the kernel designed to restrict a variety of attack avenues
- Most important for our discussion (and the most talked about feature) is the non-executable stack
Openwall Kernel Patches

- **Non-executable stack**
  - Similar to that included in Solaris and HP-UX
    - Solaris (since 2.6) `noexec_user_stack` kernel parameter
    - HPUX (since 11i) `executable_stack` kernel parameter
  - The CPU will refuse to execute code located in the stack region
    - Programs that attempt to do so will receive a Segmentation Violation
    - Logged to syslog
Openwall Kernel Patches

Program Image

Code

... printf("Hello!\n"); ...

PLT

printf() Stub

Stack

Fault

User Buffer
Openwall Kernel Patches

- Some programs normally attempt to execute code on the stack
  - Debuggers (inject code into the debugged process)
  - GCC Trampolines (nested functions)
  - Kernel sigreturn() code
- Trampolines/Signal code can be handled automatically
  - Detect call into stack rather than ret onto stack
- Stack protection can be switched off for programs to be debugged
The process space of an executable already contains all the useful routines an attacker would like to execute.

Why bother trying to insert new code?

Given a known version of the OS and a known version of libc the location of routines in libc is absolutely predictable.
Return Into libc/code

Stack

buffer[100]
Saved Base
Return Addr
*UserString

*libc:system
  *"/bin/sh"

"/bin/sh"

libc Image

system()
Openwall Kernel Patches

- To try to combat return into libc style attacks, the Openwall patches map shared libraries on addresses containing 0x00.
- Since 0x00 is a string terminator standard, string overflows can no longer construct the address or arguments.
Return into code

- While libc address may be impossible to construct directly there are still many other targets
  - PLT is at known location, may even contain stub for useful functions like system()
  - Program image itself may contain useful code, for example su contains code to setuid() and spawn shell
Openwall Kernel Patches - Strengths

- Reduces ability of attacker to execute their own code in program buffers
- Minor CPU impact
- Other features of patches not discussed today also improve system security
Openwall Kernel Patches – Weaknesses

- Code in static buffers or on the heap may still be executed
- Return into code style attacks still possible
- Some programs validly try to execute generated code, they fail
**PaX**

http://pageexec.virtualave.net/

- Kernel patches that result in non-executable:
  - Stack
  - Heap
  - BSS (Static Storage)

- This effectively wipes out all places an attacker could get arbitrary code stored

- Basically only return into code attacks are viable
PaX

- To prevent return into code attacks PaX introduced total Address Space Layout Randomization (ASLR)
- Automatically randomizes base addresses of the stack and heap
- Also randomizes the load address of all libraries
  - Making it impossible to predict the location of libc or other libraries inside the target executable
  - Thus return into libc exploits are avoided
PaX

- The only remaining avenue for attack against PaX is return into code, most likely the PLT.
- But PaX can even be told to load the executable itself at a random address.
- This previously hasn’t been possible because executable code has expected addresses compiled in.
PaX

- To make it work PaX keeps two copies of the executable in memory,
  - One non executable (at the original location)
  - One executable (somewhere else)

- The executable one faults when it tries to jump to a location in the non executable area (with an absolute address)
  - PaX intercepts the crash and redirects to the new executable location
PaX

- PaX can have its various features switched off on a program to program basis
PaX - Strengths

- Makes arbitrary execution flow and arbitrary code execution attacks *extremely* difficult
PaX - Weaknesses

- Total address space layout randomization has a significant performance impact
  - Executable remapping is not stable
- Some programs validly try to execute generated code, they fail
Program State

- All of the attacks described provide the attacker the ability to corrupt program state
- Doesn’t *have* to result in code execution
  - Overwrite stored credentials
  - Overwrite authentication state
Program State

- All of the attacks described provide the attacker the ability to corrupt program state
- Doesn’t *have* to result in code execution
  - Overwrite stored credentials
  - Overwrite authentication state
Other Generic Protections

- **Compilation Time**
  - StackGuard & FormatGuard
    - [http://www.immunix.org](http://www.immunix.org)
  - ProPolice

- **Monitoring**
  - SysCallTrack

- **Access Restriction**
  - SysTrace
    - [http://www.citi.umich.edu/u/provos/systrace/](http://www.citi.umich.edu/u/provos/systrace/)
Why aren’t they widely deployed?

- Fear of impact on production systems
- Unsupported software
- Compatibility

However… in the vast majority of cases these approaches provide a strong “immune” system with little adverse impact.
Biodiversity

- In nature a “virus” that has survived the immune system hasn’t won just yet
  - The conditions in the host must still be favourable
  - If the host behaves differently from conditions the “virus” expects, infection may be avoided
Biodiversity

- Unfortunately most operating environment installations are extremely predictable
  - Homogenous pre-compiled binary software
  - Commodity hardware
- If your installation behaves differently, attacks can be confused and stifled
  - Normally only helps against remote attacks
Biodiversity

- Simply recompiling all the applications you have access to the source to will help
  - Make addresses less guessable (particularly if you use the latest compiler with special options)
- But it’s also possible to modify the program execution environment in more general ways
  - We’ll discuss a couple, many more are possible
  - Not designed to hold back the tide, just be different
Moving Libraries

- Libraries are not mapped at fixed addresses
  - But they are highly predictable
- Each library is mapped from a known base, one by one moving upwards in the process image
- A remote attacker that knows the operating environment version of the target knows where each library (and each routine) will be
Moving Libraries

libc Image

Offsets
0x0
0x40ae0
0x91c90
0x91d48

0x0
0x42e54
0x94004
0x940bc

fork()
execve()
system()
Moving Library Images

- We can in fact move the image of a library, padding it’s start as much as we like
- So that the routines are no longer at the predictable location.
- An attacker can still determine the address but will have to brute force it
  - Depending how many bytes of padding are added, this could take a while
Moving Library Bases

- We can also move the base of the library by forcing a padding library into every process.
- All we need to do is create an empty shared library which will take up some space and force it in via:
  - LD_PRELOAD
  - /etc/ld.so.preload
  - addlibrary
Moving Library Bases

Process Library Space

0x40000000
ld-linux.so
0x4001a000
libc.so
0x4001e000
libtermcap.so

0x40000000
ld-linux.so
0x4001a000
libempty.so
0x4002d000
libc.so
0x40031000
libtermcap.so
Moving the Stack Base

- The stack starts at a fixed location and grows down (on most architectures/OSs)
  - 0xc0000000 on Linux
- Stack smashes usually rely on being able to know (or brute force) the address of data on the stack
- We can trivially move the stack with a preloaded library
Moving the Stack Base

- Does not help for most format string attacks
  - They often give away stack information
- Does not help for stack locations of data constructed during program load
  - Program Arguments
  - Environment Variables
Switching Syscalls

- All privileged operations (i.e., operations that affect anything external to the process) are performed for the kernel on behalf of the process.
- The application requests the operation by performing a system call.
- System calls are called by number.
Switching Syscalls

- Syscall numbers are usually fixed and known
  - fork = 2
  - execve = 11
- Applications should not make system calls directly
  - They’re system/platform/version specific
  - libc wraps them in functions
Switching Syscalls

- In a fully dynamically linked environment libc should be the only userland code with syscall interfaces.
- Code injected by attackers will normally try to call syscalls directly.
- If we use a kernel module to switch syscalls around and patch libc the system operates but injected code will fail.
Future Generic Protections

- Program Shepherding
  

- Program characterization and sandboxing with
  - Subterfuge
  - Janus
  - SysTrace
Thanks for listening!

- Questions?
- Email: shaun@securereality.com.au