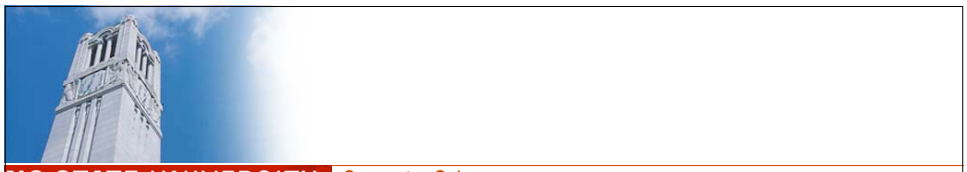


NC STATE UNIVERSITY Computer Science

CSC 405
Introduction to Computer Security

Topic 3. Program Security -- Part II

Targeted Malicious Code

- General purpose malicious code
  - Affect users and machines indiscriminately
- Targeted malicious code
  - Written for a particular system, for a particular application, and for a particular purpose
  - Trapdoor
  - Salami attack
  - Covert channel

Salami Attack

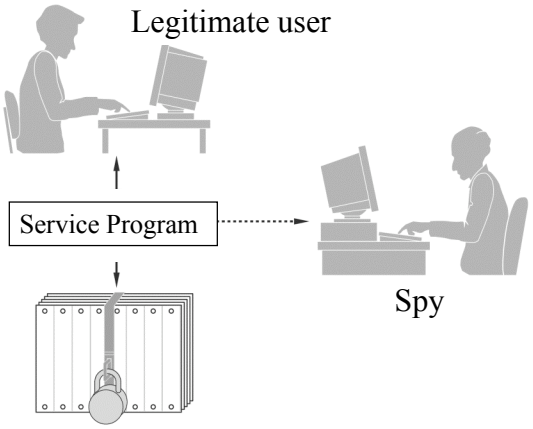
- A salami attack merges seemingly inconsequential data to yield powerful results
- Example
  - Bank programmer: transfer one cent of interest from each account to his/her account

Covert Channels

- Covert channels
  - Programs that communicate information to people who should not receive it
  - The communication travels unnoticed, accompanying other, perfectly proper, communications
- A human example
  - Reveal answers to multiple choice questions
    - Coughing for (a)
    - Sighing for (b)
    - …

Covert Channels (Cont'd)

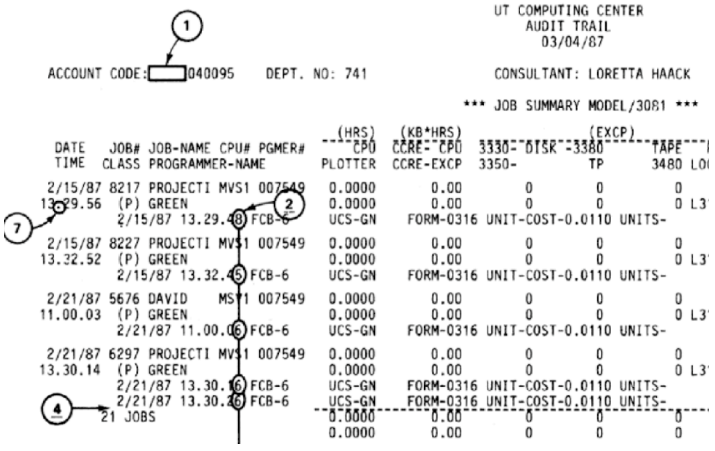
- Structure of cover channels
  - A sender
    - Service program
  - A receiver
    - Spy
  - A Trojan horse is always involved



Pfleeger/Pfleeger Fig. 03-11

How to Create Covert Channels

Example: A printed report

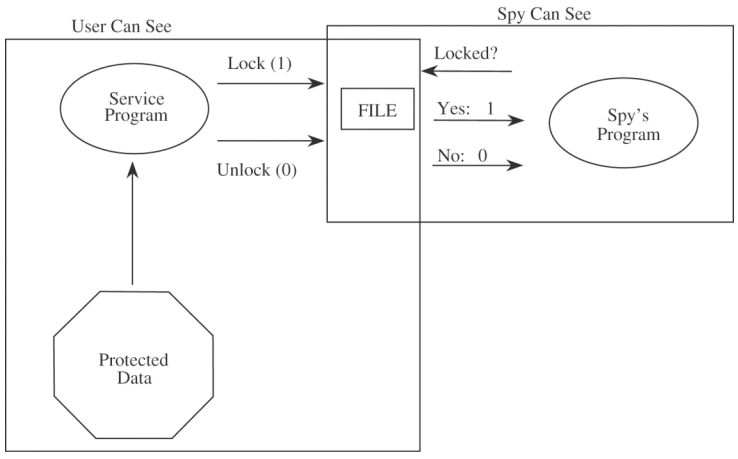


Classification of Covert Channels

- Storage channels
  - Pass information by using the presence or absence of objects in storage
- Timing channels
  - Pass information by using the speed at which things happen

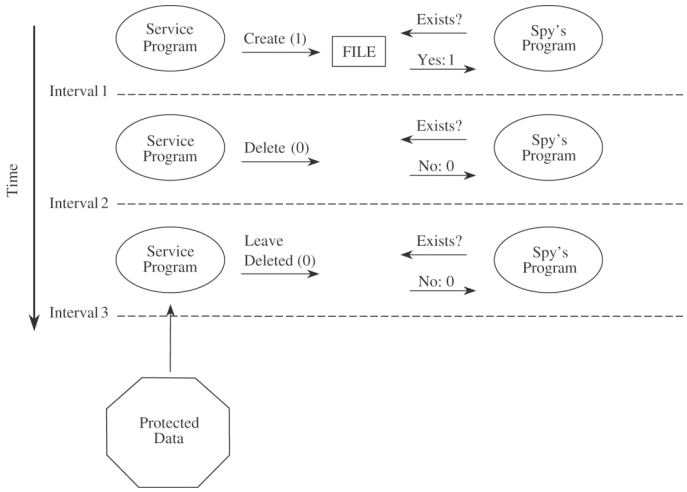
Storage Channels

- Example 1: File lock channel



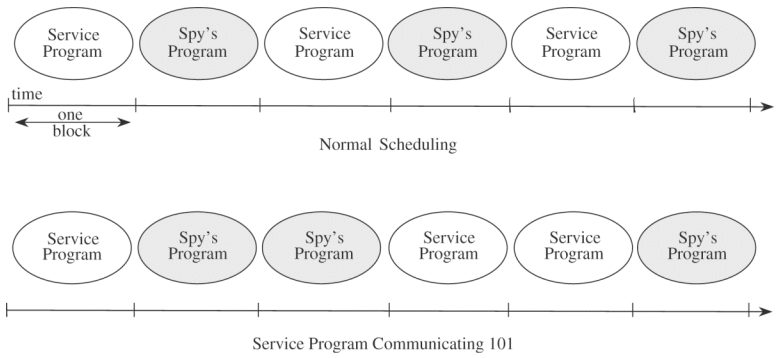
Storage Channels (Cont'd)

- Example 2: File existence channel



Timing Channels

- Example: Cover timing channel



The attacker may use error correction codes to reduce the interference from other processes.

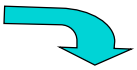
Covert Channels (Cont'd)

- The service program and the spy need access to a shared resource
  - Storage channels
    - Object in shared storage medium
  - Timing channels
    - Time

Identifying Potential Covert Channels

- Shared resource matrix
  - Basis of cover channel is a shared resource
  - ⇒ Find all shared resources
  - ⇒ Determine which processes can write to and read from the resources
  - Can be automated

	Service Process	Spy's Process
Locked	R, M	R, M
Confidential data	R	



	Service Process	Spy's Process
Locked	R, M	R, M
Confidential data	R	R

R: Can read
M: Can write

Identifying Potential Covert Channels (Cont'd)

- Information flow method
  - Static analysis of program source code
  - Explicit flow
    - B:=A
      - Information flows from A to B
    - B:=A; C:=B
      - Information flows from A to C (by way of B)
  - Implicit flow
    - IF D=1 THEN B:=A
      - Information flows explicitly from A to B
      - Information flows implicitly from D to B

Information Flow Method (Cont'd)

- Functions
  - B:=fntl(args)
    - At a superficial level, information flows from args to B
    - Need to analyze the definition of fntl
      - Information flows from global variables to B
- Need to put all pieces together to show which output are affected by which inputs
  - Can be automated

Controls against Program Security Threats

- Development controls
- Operating system controls
- Administrative controls

Development Controls

- Peer reviews
  - Review
    - Presented informally to a team of reviewers
    - Goal: consensus and buy-in before development proceeds further
  - Walk-through
    - Presented to the team by its creator
    - Goal: education; focus is on learning about a single document
  - Inspection
    - Formal process; detailed analysis in which the artifact is checked against a prepared listed of concerns
    - Goal: verify properties of the artifact of concern

Peer Review (Cont'd)

- Review log analysis
  - Do particular reviewers need training?
  - Root cause analysis ==> What should be done to discover the fault earlier?
  - Build a checklist for future reviews

Development Controls (Cont'd)

- Hazard analysis
  - Intended to expose potentially hazardous system states
  - Involves developing
    - Hazard lists, and
    - Procedures for exploring "what if" scenarios to trigger consideration of non-obvious hazards
- Example: Failure modes and effects analysis (FMEA)
    - Bottom up technique
    - Identify each component's possible faults
    - Determine what could trigger the fault and the system-wide effects of the fault
    - Often lead to possible system failures that are not made visible by other analytical means

Development Controls (Cont'd)

- Testing
  - Involves several stages
  - Unit testing
    - Each component is tested on its own, isolated from the other components in the system
    - Done in a controlled environment
    - AKA, module testing, component testing
  - Integration testing
    - Ensure the interface among the components are defined and handled properly
    - Verify that the system components work together as specified

Testing (Cont'd)

- Function test
  - Evaluate the system to determine whether the *functions* described by the requirement specification are actually performed by the system
- Performance test
  - Compare the system with the remainder of the software and hardware requirements
- Security requirements are examined during the function and performance tests

Testing (Cont'd)

- Acceptance test
  - Ensure that the system works according to customer expectations
- Installation test
  - A final test to ensure the system still functions as it should
- Regression test
  - After a change is made to enhance or fix the system, regression testing ensures that all remaining functions are still working

Development Controls (Cont'd)

- Good design
  - Using a philosophy of fault tolerance
    - Active fault detection
    - Redundancy
    - Isolate the damage and minimize the disruption
  - Having a consistent policy for handling failures
  - Capturing the design rationale and history
  - Using design patterns

Development Controls (Cont'd)

- Prediction
  - Identify what unwanted events might occur
  - Make plans to avoid them or mitigate their effects
- Static analysis
  - Examine design and code to locate and repair security flaws
    - Control flow
    - Data flow
    - Data structure
  - Many approaches; automated tools needed

Development Controls (Cont'd)

- Configuration management
  - The process by which we control the changes during development and maintenance
  - Four activities
    - Configuration identification
      - Build an inventory (baseline) of all components of the system
    - Configuration control and change management
      - Coordinate separate, related versions
    - Configuration auditing
      - Confirms that the baseline is complete and accurate, changes are recorded, recorded changes are made, the actual software is reflected accurately in the documents
    - Status accounting
      - Record the information about the components

Operating System (OS) Controls

- Trusted software
  - A part of the OS that has been rigorously developed and analyzed
  - Called Trusted Computer Base (TCB)
- Key characteristics during rigorous analysis and testing
  - Functional correctness
  - Enforcement of integrity
  - Limited privilege
    - Access is minimized; sensitive data not disclosed
  - Appropriate confidence level
- Often used as a safe way for general users to access sensitive data

Operating System (OS) Controls (Cont'd)

- Mutual suspicion
  - Programs do not trust each other
- Confinement
  - Program is strictly limited in what system resources it can access
- Access (audit) log
  - List of who accessed what objects
  - Allow tracking down what has been done

Administrative Controls

- Standards of program development
  - Capture the wisdom from previous projects
  - Standards of design
    - Design tools, languages, methodologies
  - Standards of documentation, language, and coding style
    - Layout of code, choices of variable names, recognized program structures
  - Standards of programming
    - Mandatory peer reviews, periodic code audits, compliance with standards
  - Standards of testing
    - Program verification, archiving test results, independent testers,…
  - Standards of configuration management

Administrative Controls (Cont'd)

- Separation of duties
  - Break development tasks into pieces to be performed by separate developers/testers/administrators
  - Force developers/testers/administrators to cooperate
  - More rigorous examination

Preventing Buffer Overflow Attacks

- Non-executable stack
- Static source code analysis
- Run time checking: StackGuard, Libsafe, SafeC, (Purify)
- Randomization
- Type safe languages (Java, ML)
  - Legacy code?
- Detection deviation of program behavior
- Many more …

Marking Stack as Non-Executable

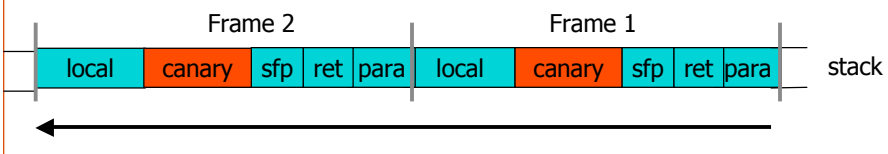
- Basic stack exploit can be prevented by marking stack segment as non-executable
  - Support in SP2. Code patches exist for Linux, Solaris
- Problems:
  - Does not defend against `return-to-libc' exploit
  - Some apps need executable stack (e.g. LISP interpreters)
  - Does not block more general overflow exploits:
    - Overflow on heap: overflow buffer next to func pointer

Static Source Code Analysis

- Statically check source code to detect buffer overflows
  - Several consulting companies
- Can we automate the review process?
- Several tools exist:
  - Coverity (Engler et al.): Test trust inconsistency
  - Microsoft program analysis group:
    - PREfix: looks for fixed set of bugs (e.g. null ptr ref)
    - PREfast: local analysis to find idioms for prog errors
  - Berkeley: Wagner, et al. Test constraint violations
- Find lots of bugs, but not all

Run Time Checking: StackGuard

- Many many run-time checking techniques …
- Solutions 1: StackGuard (WireX)
  - Run time tests for stack integrity
  - Embed "canaries" in stack frames and verify their integrity prior to function return



Canary Types

- Random canary:
  - Choose random string at program startup
  - Insert canary string into every stack frame
  - Verify canary before returning from function
  - To corrupt random canary, attacker must learn current random string
- Terminator canary:

  Canary = 0, newline, linefeed, EOF
  - String functions will not copy beyond terminator
  - Hence, attacker cannot use string functions to corrupt stack

StackGuard (Cont'd)

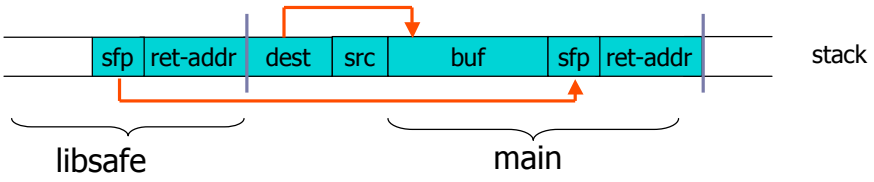
- StackGuard implemented as a GCC patch
  - Program must be recompiled
  - Minimal performance effects: 8% for Apache
- Newer version: PointGuard
  - Protects function pointers and setjmp buffers by placing canaries next to them
  - More noticeable performance effects
- Note: Canaries don't offer fullproof protection
  - Some stack smashing attacks can leave canaries untouched

Windows XP SP2 /GS

- Non executable stack
- Compiler /GS option:
  - Combination of ProPolice and Random canary
  - Triggers UnHandledException in case of Canary mismatch to shutdown process
- Litchfield vulnerability report
  - Overflow overwrites exception handler
  - Redirects exception to attack code

Run Time Checking: Libsafe

- Solutions 2: Libsafe (Avaya Labs)
  - Dynamically loaded library
  - Intercepts calls to strcpy (dest, src)
    - Validates sufficient space in current stack frame:
      |frame-pointer – dest| > strlen(src)
    - If so, does strcpy
    - Otherwise, terminates application



More Methods …

- StackShield
  - At function prologue, copy return address RET and SFP to "safe" location (beginning of data segment)
  - Upon return, check that RET and SFP is equal to copy
  - Implemented as assembler file processor (GCC)

Randomization: Motivation

- Buffer overflow and return-to-libc exploits need to know the (virtual) address to which pass control
  - Address of attack code in the buffer
  - Address of a standard kernel library routine
- Same address is used on many machines
  - Slammer infected 75,000 MS-SQL servers using same code on every machine
- Idea: introduce artificial diversity
  - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine

Randomization

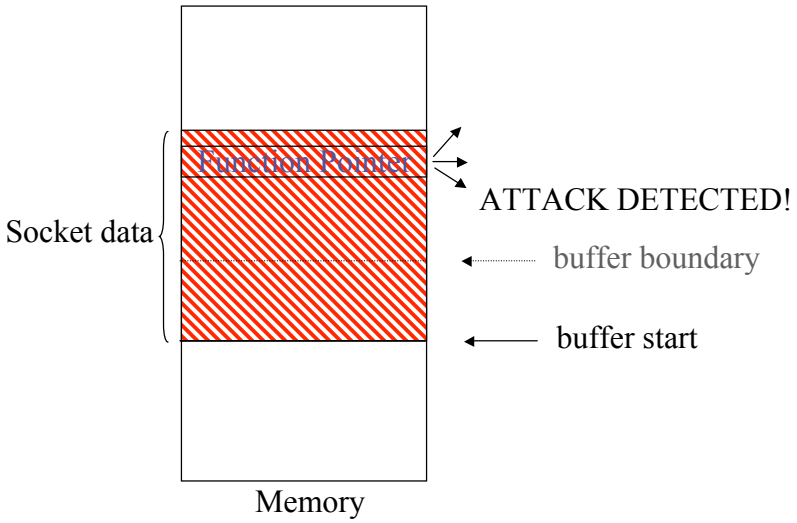
- PaX Address Space Layout Randomization
  - Randomize location of libc
  - Attacker cannot jump directly to exec function.
  - Attacks:
    - Repetitively guess randomized address
    - Spraying injected attack code
- Instruction Set Randomization (ISR)
  - Each program has a *different* and *secret* instruction set
  - Use translator to randomize instructions at load-time
  - Attacker cannot execute its own code.

Dynamic Taint Analysis

- Hard to tell if data is sensitive when it is written
  - Binary has no type information
- Easy to tell it is sensitive when it is used
- Dynamic Taint Analysis:
  - Keep track of tainted data from untrusted sources
  - Detect when tainted data is used in a sensitive way
    - e.g., as return address or function pointer

Reference:
James Newsome and Dawn Song. "Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software," In *Proceedings of Network and Distributed Systems Security Symposium*, Feb 2005.

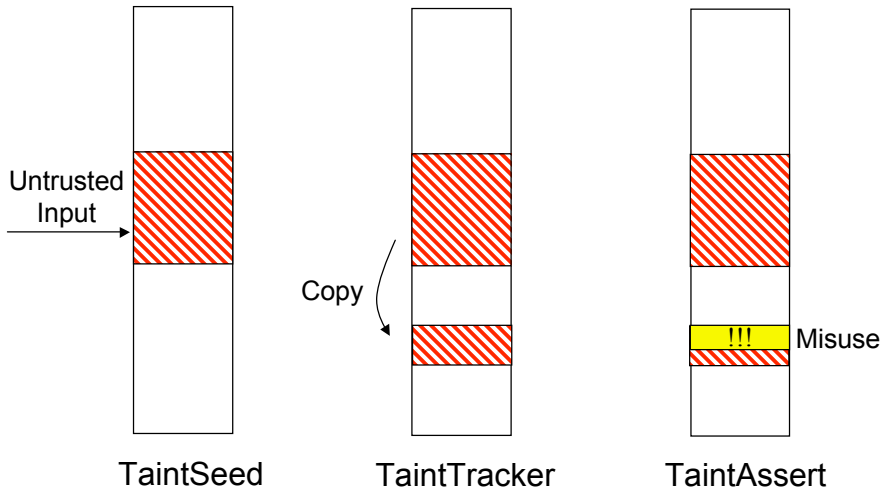
Example: Detecting a Buffer Overflow



Design & Implementation: TaintCheck

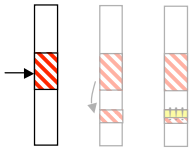
- Use Valgrind to monitor execution
  - Instrument program binary at run-time
  - No source code required
- Track a taint value for each location:
  - Each byte of tainted memory
  - Each register

TaintCheck Components



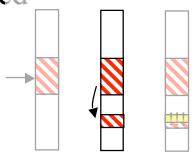
TaintSeed

- Monitors input via system calls
- Marks data from untrusted inputs as tainted
  - Network sockets (default)
  - Standard input
  - File input
    - (except files owned by root, such as system libraries)



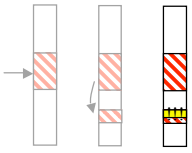
TaintTracker

- Propagates taint
- Data movement instructions:
  - *e.g.*, move, load, store, etc.
  - Destination tainted iff source is tainted
  - Taint data loaded via tainted index
    - *e.g.,* unicode = translation_table[tainted_ascii]
- Arithmetic instructions:
  - *e.g.,* add, xor, mult, etc.
  - Destination tainted iff *any* operand is tainted
- Untaint result of constant functions
  - xor eax, eax

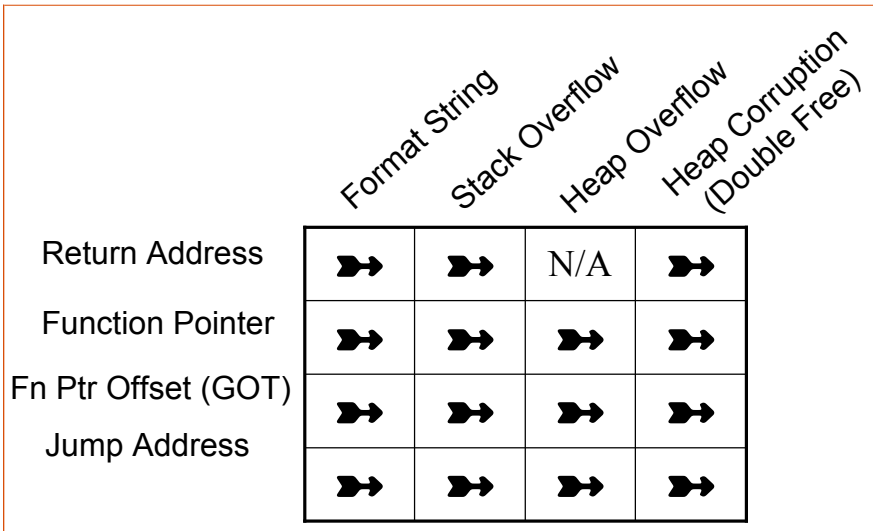


TaintAssert

- Detects when tainted data is misused
  - Destination address for control flow (default)
  - Format string (default)
  - Argument to particular system calls (e.g., `execve`)
- Invoke Exploit Analyzer when exploit detected



Coverage: Attack Classes Detected



Vigilante

- Automates worm defense
- 'Collaborative Infrastructure' to detect worms
- Negligible rate of false positives
- Network-level approaches do not have access to vulnerability specifics

Reference:
Manuel Costa, Jon Crowcroft, Miguel Castro, Anthony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham, "Vigilante: End-to-End Containment of Internet Worms", *SOSP'05*, Brighton, UK, October 2005.

Solution Overview

- Run heavily instrumented versions of software on honeypot or detector machines
- Broadcast exploit descriptions to regular machines
- Generate message filters at regular machines to block worm traffic
- Requires separate detection infrastructure for each particular service

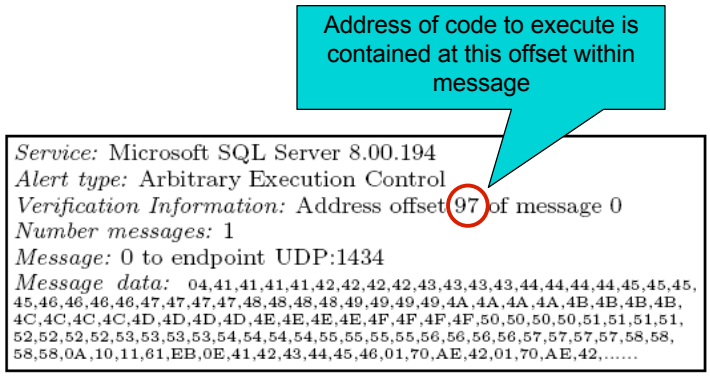
SCA: Self-Certifying Alert

- Allows exploits to be described, shipped, and *reproduced*
- Self-Certifying: to verify authenticity, just execute within sandbox
- Expressiveness: Concise or Inadequate?
  - Worms defined as 'exploiters of vulnerability' rather than 'generators of traffic'

Types of Vulnerabilities

- *Arbitrary Execution Control*: message contains address of code to execute
- *Arbitrary Code Execution*: message contains code to execute
- *Arbitrary Function Argument*: changing arguments to 'critical' functions. e.g exec
- How about others?

Example SCA: Slammer



Alert Generation

- Many existing approaches
  - Non-executable pages: faster, does not catch function argument exploit
  - Dynamic Data-flow Analysis: track *dirty* data
    - Basic Idea: Do not allow incoming messages to execute or cause arbitrary execution

Alert Verification

- Hosts run same software with identical configuration within sandbox
- Insert call to *Verified* instead of:
  - Address in execution control alerts
  - Code in code execution alerts
- Insert a reference argument value instead of argument in arbitrary function argument alert

Alert Verification

- Verification is *fast, simple and generic,* and *has no false positives*
- Assumes that address/code/argument is supplied verbatim in messages
  - Works for C/C++ buffer overflows, but what about more complex interactions within the service?
- Assumes that message replay is sufficient for exploit reproduction
  - Scheduling policies, etc?
  - Randomization?

Alert Distribution

- Flooding over secure Pastry overlay
- What about DOS?
  - Don't forward already seen or blocked SCAs
  - Forward only after Verification
  - Rate-limit SCAs from each neighbor

Local Response

- Verify SCA
- Generate *filters* – conjunctions of conditions on single messages
  - Data flow analysis: remember how dirty data propagates
  - Control flow analysis: generate filter condition to remember under what condition the vulnerability is exploited
- Two levels : general filter with false positives + specific filter with no false positives
  - General filter: specific filter with some conditions removed
    - Bytes after the vulnerable offset
    - Condition introduced by function calls