Lecture 7

Protection and Security

(Textbook Chapter 15)

Protection

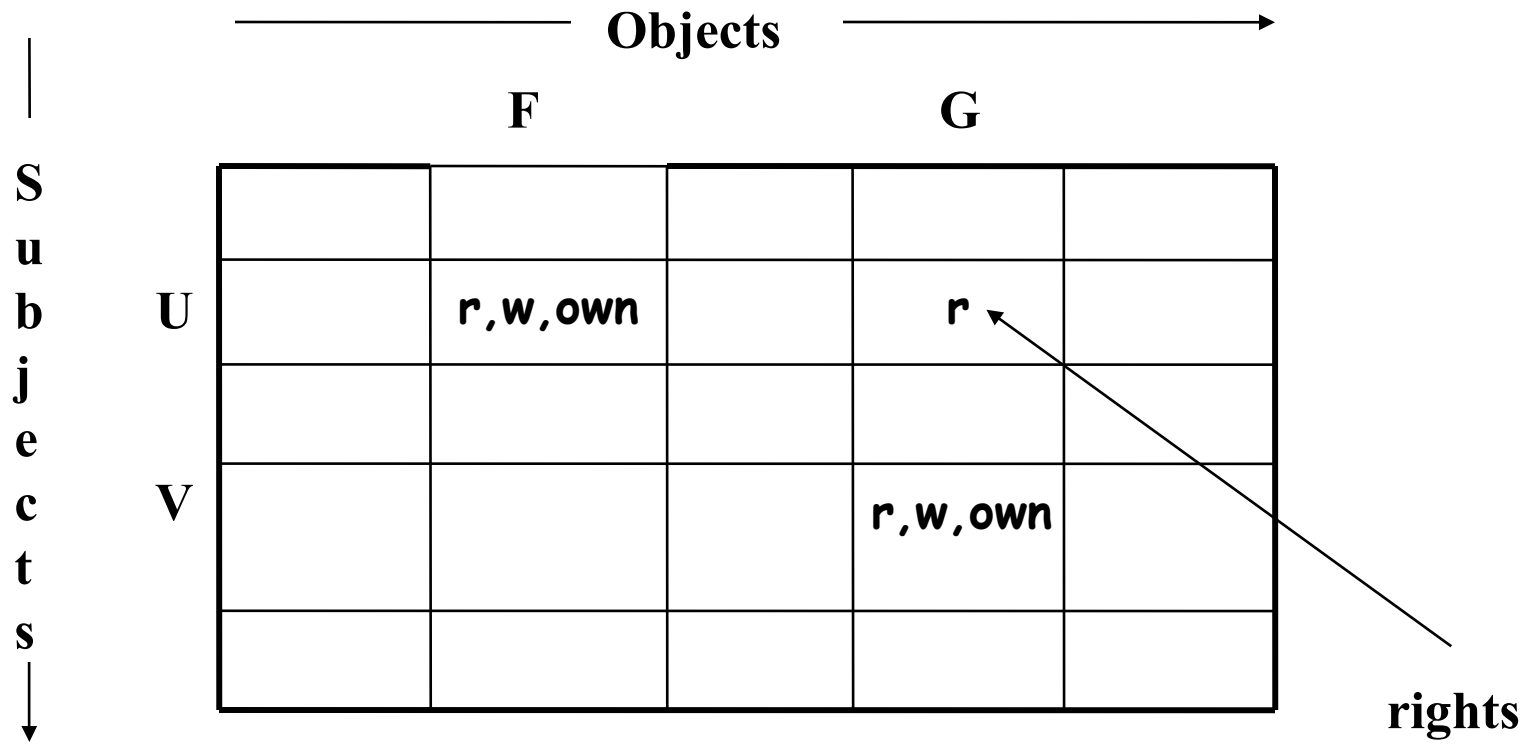
- We have talked briefly about protection in the context of file management
- Access Control
  - Conceptual model: Access Matrix
  - Reality
    - Access Control Lists
    - Capabilities

Any specific protection mechanisms in commodity OSes?

Access Matrix Model

- Basic Abstractions →
  - Subjects: A subject is a program (application) executing on behalf of a user
  - Objects: An object is anything on which a subject can perform operations (mediated by rights)
  - Rights
- Subjects and objects are relative terms
  - A subject in one context can be an object in another context
  - Example:
    - A process is a subject when it reads a file
    - The same process is an object when another process kills it.

Access Matrix Model (Cont'd)



Implementation of Access Matrix Model

- Access Control Lists
- Capabilities
- Relations

Access Control List

- ACLs
  - Store the column of the access matrix along with the objects.
  - Associated with each object is a list of all domains or principals (aka subjects) that have access to the object and the type of access allowed. Default rights are given for domains not listed. Access is permitted if at least one entry in the ACL allows it.
  - If too many domains exist, they can be grouped into classes. The Unix group id is an example of domain class.
  - Is it easy to revoke an access right? →

Access Control List

- ACLs
  - Store the column of the access matrix along with the objects.
  - Associated with each object is a list of all domains or principals (aka subjects) that have access to the object and the type of access allowed. Default rights are given for domains not listed. Access is permitted if at least one entry in the ACL allows it.
  - If too many domains exist, they can be grouped into classes. The Unix group id is an example of domain class.
  - Is it easy to revoke an access right? →
    Only the object may modify the access list, i.e. protection is implemented by the object, so revocation of rights is easy.

Example: Unix

- Each subject (process) and object (file, socket, etc) has a user id. Each object also has a group id and each subject has one or more group ids.
- Objects have access control lists that specify read, write, and execute permissions for user, group, and world.
- A subject can r,w,x an object if: →
  - (1) it has the same uid and the appropriate user permission is set;
  - (2) one of its gid's is the same as that of the object and the appropriate group permission is set; or
  - (3) the appropriate world permission is set.
  - Exceptions? →

Example: Unix

- Each subject (process) and object (file, socket, etc) has a user id. Each object also has a group id and each subject has one or more group ids.
- Objects have access control lists that specify read, write, and execute permissions for user, group, and world.
- A subject can r,w,x an object if: →
  - (1) it has the same uid and the appropriate user permission is set;
  - (2) one of its gid's is the same as that of the object and the appropriate group permission is set; or
  - (3) the appropriate world permission is set.
  - Exceptions? →
    Super-users (uid root) can do anything.

Capabilities

- Capabilities
  - For each subject (or each group of subjects): set of objects that can be accessed + access rights.
  - Problem: potentially long lists of capabilities, especially for super-users.
  - To avoid tempering w/ capabilities by a subject (process), capabilities must be protected from access: →
    - tagged architecture -- capabilities have a tag bit and can only be modified in kernel mode
    - kernel space capabilities -- capabilities are stored in the kernel and can only be accessed indirectly (e.g. file descriptors in Unix)
    - encrypted capabilities – can be stored in user space

Capabilities (Cont'd) →

- Capabilities: object-specific rights (read, write, execute) + generic rights (limit distribution and/or copy/deletion of capabilities)
- Is it easy to revoke an access right? →

Capabilities (Cont'd) →

- Capabilities: object-specific rights (read, write, execute) + generic rights (limit distribution and/or copy/deletion of capabilities)
- Is it easy to revoke an access right? →
  Revocation of rights is hard. One solution: Use version numbers incremented by objects and reject capabilities with older versions

Access Control Triples (Relations)

Subject	Access	Object
U	R	F
U	W	F
U	Own	F
U	R	G
V	R	G
V	W	G
V	Own	G

Usually in Database Management Systems.

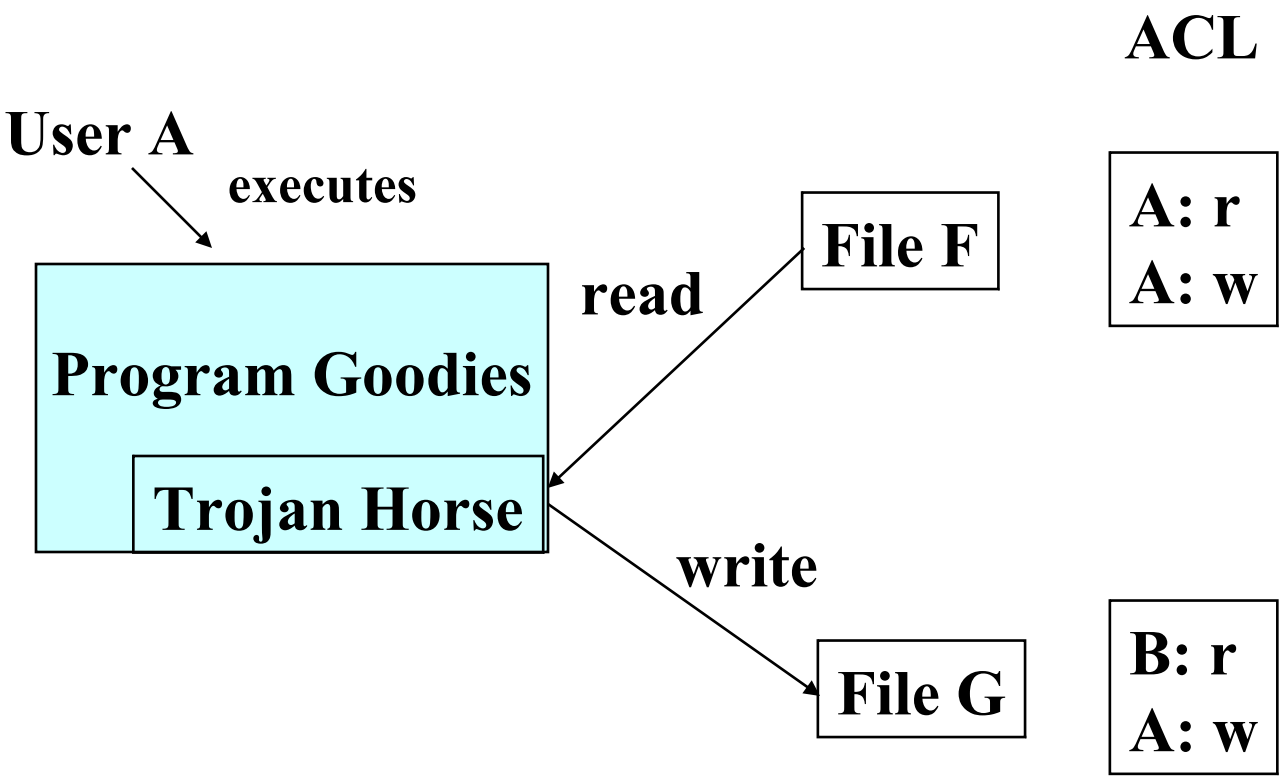
Discretionary Access Control v.s. Mandatory Access Control

- Discretionary Access Control (DAC)
  - Restrict the access of subjects to objects on the basis of identity of the requestor and access rules.
  - Allow access rights to be propagated from one subject to another. (Possession of an access right by a subject is sufficient to allow access to the object.)
- Mandatory Access Control (MAC)
  - Restrict the access of subjects to objects on the basis of security labels.

Inherent Weakness of DAC

- Unrestricted DAC allows information from an object which can be written to any other object which can be read by a subject.
- The user can be trusted not to do this deliberately. However, it is still possible for Trojan Horse Programs to do so.
  - A Trojan horse does what a user expects it to do, but in addition exploits the user's legitimate privilege to cause a security breach.

Trojan Horse Example



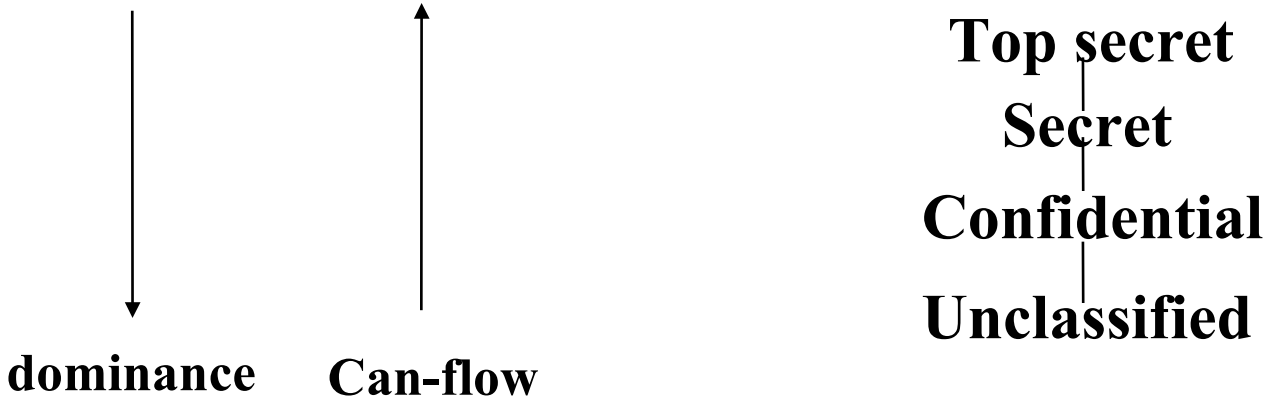
The ACLs do not allow B to read F. But B can read the information with the help of the Trojan Horse.

Mandatory Access Control

- Basic idea: put restrictions on access rights.
  - Label both the subjects and the objects.
  - Allow a subject to access an object only when certain constraints are satisfied.

Mandatory Access Control

- Bell LaPadula (BLP) Model
  - Simple security: Subject S can read object O only if
    - Label(S) dominates label(O).
    - Information can flow from label(O) to label(S)
    - Intuitively, no read up
  - Star property: Subjects can write object O only if
    - Label(O) dominates label(S)
    - Information can flow from label(S) to label(O).
    - Intuitively, no write down.



Security

- Intruders
  - Types of intruders
  - Intrusion techniques
- Malicious software
  - Trap doors
  - Logic bombs
  - Trojan horses
  - Viruses
  - Worms
  - Zombies
- Encryption
  - Conventional
  - Public key

Any security threats to access control?

Types of Intruders

- Masquerader: an individual who is not authorized to use the system and who penetrates the system to exploit a legitimate user's account.
  - Example: a person who manages to guess the password of a legitimate user
- Misfeasor: a legitimate user who accesses data, programs, or resources for which such access is not authorized, or who is authorized for such access but misuses his or her privileges.
  - Example: a legitimate user who manages to become another user by having the latter execute a Trojan horse (details later)
- Clandestine user: an individual who seizes supervisory control of the system and uses this control to evade auditing and access controls or to suppress audit collection.
  - Example: a legitimate user who manages to become a super-user user by executing a setuid program owned by root

Intrusion Techniques

- Goal of intruders: Gain access to a system or increase privileges with the system. This requires the intruder to acquire information that should be protected. Mostly, this information is in the form of a user password.
- Techniques for learning passwords: →
  - Guess the password (brute force or informed)
  - Use Trojan horse (described later) to bypass restrictions on access
  - Tap the line between a remote user and the host system
- Countermeasures:
  - Intelligent password selection: e.g. long passwords that are not words and have multiple types of characters (user or computer-generated)
  - Password protection: one-way (not reversible) encryption or controlled access to the password file
  - Intrusion detection: based on logs of activity. Assumes that behavior of an intruder is different than that of a legitimate user

Malicious Software

- Trojan horse: is a useful or apparently useful program or command procedure containing hidden code that, when invoked, performs some unwanted or harmful function. Example: apparently harmless game that when executed by a super-user reads a protected password file; fake login program
- Trap door: is a secret entry point into a program that allows someone who is aware of the trap door to gain access without going thru the usual security access procedures. Example: special shell command to start a Trojan horse
- Logic bomb: is code embedded in some legitimate program that is set to "explode" when certain conditions are met. Example: payroll program that checks whether some employee failed to appear in two consecutive payroll calculations and, if so, triggers the bomb

Malicious Software (Cont'd)

- Virus: is a program that can "infect" other programs by modifying them; the modification includes a copy of the virus program, which can then go on to infect other programs. Example: virus that attaches to an executable file and replicates, whenever the file is run
- Worm: is a network program that spreads from system to system. Once active within a system, a worm can act as a virus, implant a Trojan horse, or perform any number of disruptive actions. Example: a worm emails a copy of itself to other systems
- Zombie: is a program that secretly takes over another Internet-attached computer and then uses that computer to launch attacks that are difficult to trace to the zombie's creator. Example: zombie that is planted on hundreds of computers belonging to unsuspecting 3rd parties and are used to overwhelm a Web site.

Encryption

- *Conventional* (or symmetric-key or single-key) encryption has five ingredients:
  - plaintext
  - encryption algorithm: transforms the plaintext
  - secret key: the other input to the algorithm
  - ciphertext: output of algorithm
  - decryption algorithm: takes ciphertext + key and produces the original plaintext
- Requirements:
  - Strong encryption alg, i.e. opponent should be unable to decrypt the ciphertext or discover the key, even if he/she can see a number of ciphertexts and the plaintext that produced each ciphertext
  - Secret key must be communicated between sender and receiver in secure form (e.g. personally and outdoors) and kept secure

Conventional Encryption

- Types of attack
  - Cryptanalysis: relies on nature of alg and perhaps knowledge of characteristics of plaintext or even some plaintext-ciphertext pairs
  - Brute force: attempts to try every possible key on a piece of ciphertext until an intelligent translation into plaintext is obtained

Conventional Encryption (cont.)

- Example: Data Encryption Standard (DES)
  - Plaintext has to be 64 bits and key has to be 56 bits. Longer blocks of plaintext encrypted in chunks of 64 bits
  - DES algorithm: perform 16 iterations of complex transformations (permutation of bits and substitution of one bit pattern for another) on plaintext; input of each iteration is output of previous iteration plus a permutation of the key bits
  - Avg. time for brute force search of 56-bit key
    - at 1 decryption/msec (today's computer): 1142 years
    - at 1M decryptions/msec (massively parallel computers): 10 hours
  - DES broken in 1998. It will soon be replaced by a stronger standard. Still debate over what standard.

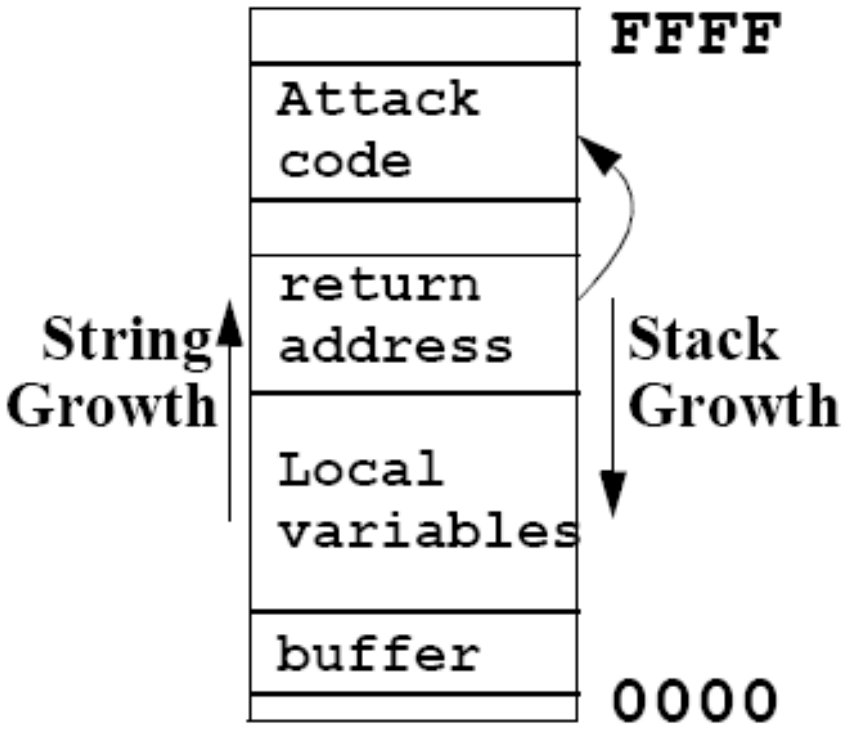
Encryption (cont.)

- *Public-key* (or asymmetric) encryption: instead of a single key, two keys – public and private keys
- Idea: Use public key of destination to encrypt the message. Send the message. Destination uses the ciphertext and the private key to decrypt the message
- Advantage: private key is not communicated, whereas public key does not have to be communicated in extremely secure form, since one would need both keys to break the encryption

Public-Key Encryption

- Example: RSA algorithm.
  - Plaintext and ciphertext are integers between 0 and n-1 for some n (a multiplication of two large prime numbers p and q)
  - Based on n, p, and q do some magic to generate the two keys
  - Only known/possible cryptanalytic attack to RSA is to factor n, which is VERY HARD! Authors of RSA claim that, if n has about 200 digits, it would take 10^23 mathematical operations to crack it. At 1 GHz this means more than 1M years!

Buffer Overflow Attack (Stack Smashing)



- Injected attack code typically spawns a shell with root privileges
- StackGuard prevents injected attack code from executing
- If program has buffer overflow vulnerability → attacker can crash Stack Guard

Marking stack non-executable

- Basic stack exploit can be prevented by marking stack segment as non-executable
  - Code patches exist for Linux and Solaris.

- Problems:
  - Some apps need/use executable stack (e.g., LISP interpreters, graphics drivers)
  - Does not block more general overflow exploits:
    - Overflow on heap: overflow buffer next to func pointer.
    - Cannot make all the data segment non-executable
      - Modern OSs may perform JITing in data area
    - Can re-enable execute right for stack/data

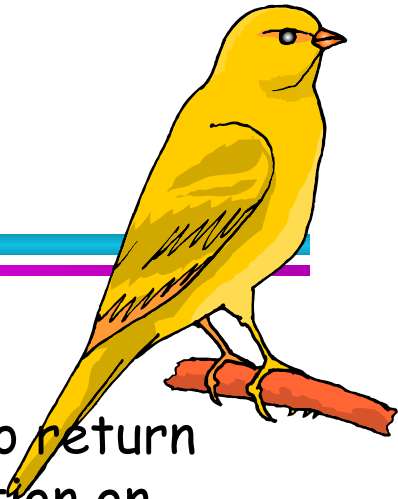
Static source code analysis

- Statically check source to detect buffer overflows.
  - Several consulting companies

- Several tools exist to automate the review process:
  - Stanford: Engler, et al. Test trust inconsistency
  - @stake.com (l0pht.com): SLINT (designed for UNIX)
  - Berkeley: Wagner, et al. Test constraint violations

- Find lots of bugs, but not all

StackGuard



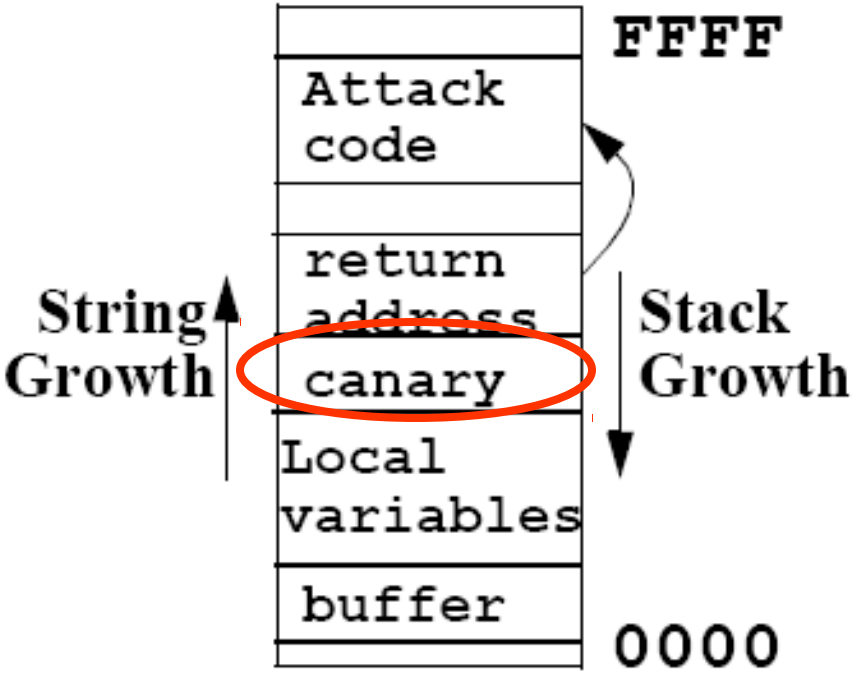


Figure 2: StackGuard Defense Against Stack Smashing Attack

StackGuard

1. prevents change to return address of a function on stack
   - prevent change to address or
   - prevent write to return address
2. effective when return address cannot be altered → more overhead
   - supports both modes

How StackGuard is Implemented

- return address is unaltered **iff** canary word is unaltered
- Only really works for buffer overflow attacks:
  - Write attacks assumed to be sequential
  - Unsafe if canary can be guessed
- StackGuard implementation:
  - modified gcc compiler: function_prologue / function_epilogue
  - canary word added to stack
  - checks if canary unmodified before function return
  - < 100 extra lines of code in gcc
  - Canaries need to be random
  - Up to 8% overhead (Apache)

MemGuard: Preventing return address changes

- MemGuard protects return address → sandbox
  - make it read-only when function is called
  - un-protect return address when function finishes and returns
- protection / un-protection in gcc's function_prologue /"_epilogue
- Higher overhead than StackGuard but more secure
- Immunix project solution: Run StackGuard in Canary version until a treat is detected, then run with MemGuard
- Now Novell's AppArmor product for Linux
  - Others: random canaries in Visual Studio 2003…
- Others:
  - dyn. bounds checking (Purify, libsafe…) ->higher overhead
  - Address obfuscation (XOR on re add) → only pseudo-random
  - Address randomization (stack base, heap, …)

Timing attacks: example

- Consider following pwd checking code:

```
int password-check( char *inp, char *pwd)
    if   (strlen(inp)  !=  strlen(pwd)) return 0;
    for( i=0; i < strlen(pwd); ++i)
        if  ( *inp[i]  !=  *pwd[i]  )
            return 0;
    return 1;
```

- Simple timing attack exposes password one character at a time
- Threat to Smartcards, Cell phones, …