

# Course 1: Overview of Secure Programming, Section 5

Pascal Meunier, Ph.D., M.Sc., CISSP

May 2004; updated January 12, 2006

Developed thanks to support and contributions from Symantec Corporation, support from the NSF SFS Capacity Building Program (Award Number 0113725) and the Purdue e-Enterprise Center

Copyright (2004) Purdue Research Foundation. All rights reserved.



## Course 1 Learning Plan

- Security overview and patching
- Public vulnerability databases and resources
- Secure software engineering
- Security assessment and testing
- **Shell and environment**
- Resource management
- Trust management

## Learning objectives

- Understand how shells interpret commands, launch and provide environments for processes
  - Understand how setuid or LocalSystem scripts and programs are risky
- Understand how environments affect the security of applications
- Understand how configuration issues affect the security of applications
- Understand security issues in audit logs
- Understand security issues in launching processes

# Operations Management and Best Practices

- **Shells**
- Environment
- Configuration
- Logging
- Calling External Programs

## Shells: Outline

- What is a shell?
- Relative path vulnerabilities and mini-lab
- Substitutions
- Setuid scripts and programs

# What is a shell?

- Launches programs, including other shells
- Provides
  - Capabilities to applications
  - A user interface
- Windows Explorer shell
- Replacement Windows shells
  - Geoshell
  - Aston, etc...
  - Norton Desktop for Windows
- UNIX shells
  - bash
  - tcsh, etc...

## Capabilities provided by Windows shells

- Custom url handlers
  - Clicking on a url "outlook://" starts outlook
  - Buffer overflow when handling custom urls of improperly removed applications (CVE-2002-0070; MS02-014)
- UI preferences
  - Buffer overflow when handling desktop.ini (CVE-2003-0306; MS03-27)
- Path resolution and handling
  - Relative shell path vulnerability in Windows 2000 and NT (CVE-2000-0663)
    - ❖ Run Explorer.exe trojan from another user
- Various means of launching other programs
  - Buffer overflows (CVE-2002-1327, CVE-2003-0503)

# Capabilities of UNIX shells

- Substitutions
  - Filename substitution (wildcard expansion, a.k.a. globbing)
  - Command substitution
    - ❖ bash and tcsh interpret backticks in names (CVE-1999-1383)
    - ❖ Arbitrary command execution in GhostView handling of file names (CVE-2002-1569) (BID 5840)
    - ❖ Several other applications invoke shell capabilities!
- Environment variables
  - PATH search variable
- File system path resolution
- Launching applications

## Relative Path Vulnerabilities

- Relative paths trigger a search for the actual file. Is it:
  - In the current directory?
  - In some other directory specified by the PATH environment variable?
  - Which one of the above two should be done first?
    - ❖ Insecure default in Windows (see next slide)
- ICAT: 15 entries (Windows, UNIX, others)
- Common misconfiguration of UNIX accounts
- Mini-Lab

# Windows PATH

- Different behavior depending on version of Windows
  - Old behavior: current directory was searched first for DLLs
  - New behavior: search all system locations first
    - ❖ XP SP1
    - ❖ Windows 2003
  - More secure, but...
- The current directory is searched even if "." is not in your PATH, and searched *before* your PATH!
  - Insecure default
  - You may not get the DLL (dynamically loaded library) you wanted! (see Howard and Leblanc 2003)

# Windows Filename Extensions

- What if you didn't specify the extension?
- The environment variable PATHEXT decides the order (.com, .exe, .bat, .cmd, ...)
- What if PATHEXT is changed by a malicious user, so a trojan would run instead?
- Other ambiguities
  - Trailing dot, slash in filename
  - Long vs short name
  - Canonicalization issues covered in course 2, part 7 "Canonicalization and Directory Traversal"
- Solution: Use the absolute path and complete name

## Shell Mini-Lab (Windows)

- Open a command prompt
  - Type “command” within the window you opened. Which shell is running now? Now type “exit”
  - Type “cd” or “set %CD%”. What is the current directory?
  - Type “set PATH”. What is the meaning of the output?
  - Create a cmd file named “cmd.cmd”:
    - ❖ `echo @echo gotcha > cmd.cmd`
  - Compare the execution of “cmd” and “. \cmd”. What is the difference, if any?
  - Type “set PATH=%PATH%; .”. What effect does it have when you run “cmd”?
  - Create a batch file named “cmd.bat”:
    - ❖ `echo @echo hello > cmd.bat`

## Shell Mini-Lab (Windows continued)

- What happens when you run “cmd” now?
  - How can you change the behavior?
- Compare the results of running “cmd” with the results of running “%SYSTEMROOT%\system32\cmd”.
- What kind of path is “%SYSTEMROOT%\system32\cmd”?
- Type “%SYSTEMDRIVE%”
- Type “cd %SYSTEMROOT%\system32”
- Compare the results of running “cmd” and “. \cmd”.

## Shell Mini-Lab (UNIX)

- Get into the UNIX shell provided for the class
  - Type `/bin/sh`. Which shell is running now?
  - Type `pwd`. What is the current directory?
  - Type `echo $PATH`. What is the meaning of the output?
  - Create a script named `ls`:
    - ❖ `echo "echo gotcha" > ls`
  - Allow execution by running `chmod a+rx ls`
  - Compare the execution of `ls` and `./ls`. Why is the output different?
  - Type `PATH=./bin:/sbin:/usr/bin:/usr/sbin`. What is the effect when you run `ls`?

## Shell Mini-Lab (UNIX continued)

- Compare the execution of "ls" with "/bin/ls".
- What kind of path is "/bin/ls"?
- Type "cd /bin"
- Compare the results of running "ls" and "./ls".

## Question

- A "relative path" is relative to:
  - a) Your home directory
  - b) The current working directory
  - c) The root (top) directory (e.g. "C:\\" or "/")

## Question

- A "relative path" is relative to:
  - a) Your home directory
  - b) **The current working directory**
  - c) The root (top) directory (e.g. "C:\\" or "/")

## Question

- Why is the PATH environment variable important?
  - a) It specifies the order of directories in which a shell looks for a file, when a relative path has been specified
  - b) It changes the execution path within applications
  - c) You must follow your own path

## Question

- Why is the PATHEXT environment variable important?

## Question

- In a UNIX shell, when an application runs `./filename`, which file is run?
  - a) The file of the same name ("filename") in the same directory as the application
  - b) The file of the same name ("filename") in the current working directory of the application
  - c) The file of the same name ("filename") in a directory specified by the PATH environment variable

## Question

- In a Windows shell, when an application runs "filename", which file is run? Choose the best answer.
  - a) The file of the same name ("filename") in a directory specified by the PATH environment variable
  - b) The file of the same name ("filename") in the current working directory of the application
  - c) The first file in the current directory that matches the first extension in the PATHEXT environment variable
  - d) The file of the same name ("filename") in the same directory as the application

## Question

- Which is more secure to run?
  - a) ./filename or .\filename
  - b) filename
  - c) /bin/filename or  
c:\WINNT\system32\filename.exe

## Question

- Which is more secure to run?
  - a) ./filename or .\filename
  - b) filename
  - c) /bin/filename or  
c:\WINNT\system32\filename.exe**
- Comment: Because ./filename refers to the current path, it has a level of indirection that can be exploited.
  - Always specify absolute paths unless impossible.
  - Explicitly set the PATH and any other important environment variables.

## Substitutions: Outline

- What are substitutions?
- Vulnerability due to substitutions
- How not to patch
- White list best practice
- Mini-Lab 2: A shell exploit

## What can be substituted?

- Filename substitutions (wildcards, a.k.a. globbing)
- Directory stack substitution
- Command substitution
- Subshells
- Other substitutions
  
- UNIX Example: `ls /var/*/*log*`
  - `/var/log/boot.log`
  - `/var/log/prelink.log`
  - `/var/log/Xorg.0.log`
  - `/var/run/klogd.pid`
  - `/var/run/syslogd.pid`

# GNU 'bash' prompt parsing vulnerability

## CVE-1999-0491, BID 119

- UNIX back-tick (command substitution)
  - Typing "``command``" on the command line executes the command, even if it should have been an argument for another command
- Mallory runs: `mkdir "\ `command` "`
  - Create directory with a command inside back-ticks
- Alice runs: `cd "\ `command` "`
- Mallory's command executed by Alice
  - This could happen when moving around directories with symlinks
- Code injection due to full shell interpretation of directory name

## Prompt parsing vulnerability Fixed in tcsh 6.07

- Changelog:  
“33. Add VAR\_NOGLOB, and use it to avoid globbing directory names when cd'ing into them.”
- Flag "VAR\_NOGLOB" is used to identify situations where substitution (incorrectly implied to be globbing) should not be done
- Question: Did the programmer fix *all* of the situations where substitution shouldn't be done?

# Prompt parsing vulnerability

## tcsh 6.07 Code Fix

- ```
/* set: storage into hashed/balanced tree */
void
set1(var, vec, head, flags)
    Char    *var, **vec;
    struct varent *head;
    int flags;
{
    register Char **oldv = vec;

    if ((flags & VAR_NOGLOB) == 0) {
        /* if not forbidden, do the globbing */
        gflag = 0;
        tglob(oldv);
    }
    ...
}
```

Note: double negation -- avoid if you can  
\*\*\* keep code simple if you can \*\*\*

# Prompt parsing vulnerability

## tcsh 6.07 Code Fix: "Diff" output

- diff tcsh-6.06/sh.dir.c tcsh-6.07.02/sh.dir.c

```
< set(STRdirstack, Strsave(dp->di_name), VAR_READWRITE);
```

```
---
```

```
> set(STRdirstack, Strsave(dp->di_name), VAR_READWRITE |  
VAR_NOGLOB);
```

```
< set(STRowd, Strsave(varval(STRcwd)), VAR_READWRITE);
```

```
< set(STRcwd, Strsave(dp), VAR_READWRITE);
```

```
---
```

```
> set(STRowd, Strsave(varval(STRcwd)), VAR_READWRITE |  
VAR_NOGLOB);
```

```
> set(STRcwd, Strsave(dp), VAR_READWRITE | VAR_NOGLOB);
```

# Prompt parsing vulnerability

## Discussion: What's Wrong?

- The fix forbids substitution (including globbing) only when dealing with changing directories.
- Why is this is a brittle fix and a bad practice?

# Prompt parsing vulnerability

## Discussion: Sample Answer

- The first fix forbids substitutions when printing the prompt ('\w' option)
- The second fix forbids substitution when changing directories
- The  $n^{\text{th}}$  fix forbids substitution for yet *another* situation.

## White List Best Practice

- Mechanism should *not* require enumeration of each dangerous thing (black list)
  - It's easy to miss things
- Instead, mechanism should be designed to deny all by default unless something is explicitly enumerated as safe (white list)
- Outline of incorrect fix approach:
  - Try to block or forbid a specific exploitation path
  - Result: another path is found
    - ❖ Underlying problem still exists
    - ❖ Common repeated mistake
    - ❖ Multiple patching attempts are costly and annoy customers

## Mini-Lab 2: UNIX

### An exploit to gain shell access

- Most exploits are aimed at giving an attacker *shell access*
  - "Execute arbitrary code" usually means starting a shell
- Next, a back door is installed.
- **Warning:** your account could get compromised by doing this mini-lab
  - don't perform the "**chmod**" operations (comment them out)
    - ❖ instructor will demo some student solutions in a throw-away account
  - \*or\* don't use a university account
- In this lab, you will create a back door with a setuid "C" program
  - setuid scripts are now disabled by default in many OSes

## Mini-Lab 2: UNIX

### A Back Door

- Create a C program containing:
  - `setreuid(geteuid(), geteuid())`
  - `execlp("/bin/sh", "/bin/sh", 0)`
- Do `chmod 4755 your_program` (makes it setuid)
- This is a backdoor into *your* account!
- Team up with someone else
- With their permission, execute the back door they have written
- Type "whoami" or "id" into the shell you get

## Mini-Lab 2: UNIX

### A Back-Door Creating Script

- Inside a directory where only you have execute privileges, write a shell script that will
  - Write the C program you wrote into a file
  - Compile it
  - Use `chmod` to set the setuid bit and allow anyone to run it.
- What will happen if another user runs the **shell script**?
- How can you persuade another user to run it?
- Turn in the shell script (email it to me)
- Make sure to delete the C program from your account...

## Mini-Lab 2: UNIX Answers

- What will happen if someone runs it?
  - It will create a backdoor script into their account
- How can you get someone to run it?
  - Create a directory named `bcs`
    - ❖ if using a vulnerable version of bash or tcsh
  - Send it to them as an email attachment
  - Make it do something useful or interesting, and put it in a shared area; wait for someone to run your trojan program.
  - Name it the same as a common system command and wait for someone who has "." in their path to run it by accident from a directory where you have write access.
  - Exploit a vulnerability into a program running as root or LocalSystem

## Discussion Question

- How can you configure Windows services to mitigate the consequences of a vulnerability?
- Hint: Apply the principle of least privilege

## Discussion Sample Answers

- Windows: Services that run as System
  - Change the account associated with a service ("Log On As" setting), from LocalSystem to a lower privilege account (you need to configure that account carefully, or use LocalService or NetworkService)
- Sometimes, users can't figure out why their software doesn't work so they make it run with an administrator account, which is even worse!
  - Principle of psychological acceptability: "This is too hard, so let's open it and grant it all privileges so it works!"
- UNIX: This is mitigated by configuring services to run as "nobody" or separate accounts with limited privileges for each service

## Question

- Why is a secure configuration difficult to achieve?
  - a) Operating systems are complex
  - b) Users will break security (if they can) to get their services to run
  - c) There are many services to secure
  - d) All of the above

## Question

- Why is a secure configuration difficult to achieve?
  - a) Operating systems are complex
  - b) Users will break security (if they can) to get their services to run
  - c) There are many services to secure
  - d) All of the above**
  
- Installing things secure by default is becoming more popular and expected

## Discussion Question

- What can you do to improve the security of your projects when you go back to work at the end of this tutorial?

## Environment: Outline

- What is the environment?
- Can you trust the environment?
- Environment pollution attacks

## What is the environment?

- File System
  - Correct permissions/ACLs on files
  - Partitions
- Operating System, sandboxes
- Services
- Accounts
  - Correct account permissions
- Environment variables
  - e.g., PATH
- Other defaults
  - e.g., umask (for new file default permissions)
- Logging facilities

## Trusting the Environment?

- As the developer of an application, you (should) know how to secure the environment
- Configure files with the correct permissions during installation
- Umask: Create files with correct permissions
- Environment variables are typically under the control of untrusted users

## Question

- Who controls the value of the PATH variable? What does that tell you about other environment variables? (Hint: A search for "environment" in ICAT gave 192 entries in spring 2004; the NVD now lists 310 entries as of January 17, 2006)

# Environment Pollution

- A program can get values from the environment
- Some variables are used automatically
  - Win32 process hooking, dll injection, Microsoft research `detours` library
  - UNIX `LD_LIBRARY_PATH`, `LD_PRELOAD`
    - ❖ Function interception (library interposition)
    - ❖ **Before** you get control!
- An attacker can influence environment:
  - For code injection attacks
  - To create buffer overflows
  - To bypass access controls
  - Denial of service (crashes for various reasons)

## Comments on PHP Poisoning

- PHP was designed for power and ease-of-use in CGI programming, *not* security
- CGI parameters automatically become variables within PHP scripts
  - Attacker can control logical flow of program
  - Option turned off by default in new versions of PHP
    - ❖ Used to be ON by default

## Example PHP Poisoning

- PHP: Variables inside the program can be initialized with values supplied by the remote client (register\_globals option)!
- ```
<php
if ($username == $allowed && $password
== $secret)  $authorized = "yes";
...
if ($authorized == "yes") {
...
}
?>
```
- “url?authorized=yes” bypasses authentication

## Configuration: Outline

- Default accounts
- Hard-coded passwords and backdoors
- Unsafe configuration interfaces

## Configuration Best Practice

- All dangerous configuration options should be turned off by default (SD3 – secure by design, secure by default, secure in deployment)
- The developer should know what options are dangerous
- Customer doesn't know any better so the installation and configuration program should provide guarantees
  - Exceptions should provide warnings

## Question

- Are accounts with default passwords a good secure configuration practice?
  - a) It doesn't matter, they are necessary
  - b) Yes, if they are kept secret
  - c) No
  
- Hint: see
  - <http://www.cirt.net/cgi-bin/passwd.pl>
  - <http://www.phenoelit.de/dpl/dpl.html>

## Discussion

- What can your software do instead of using accounts with default passwords?

## Discussion Sample Answers

- What can your software do instead of using accounts with default passwords?
  - Functionality could be blocked until accounts are set properly (with appropriate notices so customer doesn't think that the program or equipment is broken)
  - If the default account can't be avoided, most functionality could be disabled until the default account is removed by the administrator.

## Hard-Coded Passwords

- Open design security principle
- Hard-coded passwords are a failure in the application of that principle
  - revealed password may result in a catastrophic failure
- OEM requirements and ease-of-use may be at odds with this principle

## Wireless Access Points, Hard Drives, etc...

- Does it sound reasonably safe to you to *only* allow configuration of these devices from a wireless PC, instead of requiring a wired connection?
  - Cisco access points
    - ❖ Serial connection available for less user friendly setup
  - Other brands

# Logging

- Logging is an important, often requested security functional requirement
  - Especially if application has its own access control
- Logging integrity
  - Append-only loggers
  - External loggers
- Logging confidentiality issues (passwords)
  - Authority needed to view audit logs should be higher than the privilege to administer the product

## Attacks on Logs

- Denial of service using chaff (junk; noise)
- Semantic attacks (fake log entries)
- Log entries with special characters
- Log entries attempting to exploit vulnerabilities in the logging mechanism
- Log entries attempting to exploit vulnerabilities in the log viewing mechanism
- Logging second-hand or uncertain information
  - Spoofed identities
- Write to syslog from any location on the network

## Log Denial-of-Service Attacks

- CVE-1999-0566  
An attacker can write to syslog files from any location, causing a denial of service by filling up the logs, and hiding activities.
  - Chaff attack
  
- CVE-1999-0063  
Cisco IOS 12.0 and other versions can be crashed by malicious UDP packets to the syslog port.
  
- Solutions: Access control lists, firewall rules, etc...

## Example Logging Issues

- CVE-2003-0020  
Apache does not filter terminal escape sequences from its error logs, which could make it easier for attackers to insert those sequences into terminal emulators containing vulnerabilities related to escape sequences.
- CVE-2003-0460  
The rotatelogs program on Apache before 1.3.28, for Windows and OS/2 systems, does not properly ignore certain control characters that are received over the pipe, which could allow remote attackers to cause a denial of service.

## More Example Logging Issues

- CVE-2001-1098  
Cisco PIX firewall manager (PFM) 4.3(2)g records the `enable` password in *plaintext* to the `pfm.log` file, which could allow local users to obtain the password by reading the file.
- CVE-2002-0113  
Legato NetWorker 6.1 stores log files in the `/nsr/logs/` directory with world-readable permissions, which allows local users to read sensitive information and potentially gain privileges.

## Yet More Example Logging Issues

- CVE-2001-1403  
Bugzilla versions prior to 2.14 – username and *password* from URLs recorded in logs
- CVE-2001-0300  
oidldapd 2.1.1.1 in Oracle 8.1.7 - local users can delete logs and/or overwrite other files via a symlink attack.
- CVE-2001-0870  
HTTP server in Alchemy Eye and Alchemy Network Monitor 1.9x through 2.6.18
- CVE-2001-0908  
CITRIX Metaframe 1.8

## Logging Mistakes!

- CVE-2000-1179  
Netopia ISDN Router 650-ST allowed unauthenticated remote attackers to read system logs using certain *control characters*.
- CVE-2000-0967  
PHP 3 and 4 did not properly cleanse user-injected format strings, allowing remote attackers to execute arbitrary commands by triggering error messages that were improperly written to the error logs.
- CVE-2000-0642  
The default configuration of WebActive HTTP Server 1.00 stores the web log `active.log` in the web server's root document directory.

## Question

- Logging can be attacked by:
  - a) Exclusively using exploits that require an account on the local machine
  - b) Using maliciously constructed data
  - c) Exclusively using exploits that break an application that sends log data to the logging mechanism

## Question

- Logging can be attacked by:
  - a) Exclusively using exploits that require an account on the local machine
  - b) **Using maliciously constructed data**
  - c) Exclusively using exploits that break an application that sends log data to the logging mechanism

# Logging Defenses

- Use different logs for:
  - Events
    - ❖ For users to see why their page/project/etc... failed
    - ❖ Per user log even better
  - Operations
    - ❖ Allow normal administrative supervision
    - ❖ e.g., "started at 9:32 PM"
    - ❖ e.g., "error reading configuration file at ..."
  - Audit
    - ❖ For sensitive information
    - ❖ e.g., Failed logins
      - Usernames
      - Passwords entered in username field
        - » Both matched pairs usually available in audit log

# Calling External Programs

- Wrappers
- Calling paradigms
  - Shell
  - Special calls
    - ❖ Custom environment
- File descriptors

## The Wrapper Problem

- Calls to an insecure application can be wrapped inside and protected by another program that performs input validation and access control
- Wrapper must have an exact and complete model of the grammar and semantics used by the protected application, otherwise:
  - Insecure application loses functionality
  - Wrapper lets through dangerous (“unexpected”) commands
- Updates to an insecure application require an updated wrapper

## Example Wrapper Problem

- wu-ftp CVE-1999-0997
  - FTP server
- wu-ftp with FTP conversion enabled allows an attacker to execute commands via a malformed file name that is interpreted as an argument to the program that does the conversion, e.g. tar or uncompress.
  - Conversion is used to compress or uncompress files
  - File names passed to tar without checking if one really is a command line option that can be abused!
    - ❖ e.g., --use-compress-program <program>
    - ❖ e.g., --remove-files

## Wrapper Through Interpreted Code

- When application is invoked by a command processed through a complex interpreter
- Examples:
  - calling programs through the `system()` and `wsystem()` calls (UNIX and Windows)
  - other cases of dynamic code generation (SQL, etc...)
- Modeling the interpreter is difficult.
  - “exec” family of UNIX calls presents a simplified interface that limits misinterpretations and engineered attacks.
  - Windows "CreateProcess\*", "ShellExecute", "WinExec" have complex cases that allow engineered attacks

# Windows Call Complexity

- CreateProcess (  
    lpApplicationName,  
    lpCommandLine,  
    ...)
- If lpApplicationName is NULL, then  
lpCommandLine is interpreted...
  - Looks for an application name
    - ❖ White space separators
    - ❖ White space in directory names are "tried" as separators
      - C:\\Program Files\\ApplicationDir\\Appname.exe
      - C:\\Program.exe would get executed if present
      - Solution: enclose application in quotes, but why not specify lpApplicationName then?

## UNIX "system " Call

- Program and arguments are interpreted by a shell!
  - very difficult to model and sanitize
- Also available on Windows!
- Exec calls
  - `exec` and `execle` allow the separation of path, arguments, and environment
    - ❖ Fewer risks
  - `int exec(const char *path, char *const argv[], char *const envp[]);`

## Question

- `execlp` and `execvp` search the PATH like a shell does if the specified path is not absolute. Are they:
  - a) As safe as `execv`
  - b) Less safe than `execv`
  - c) More safe than `execv`
  
- `int execv(const char *path, char *const argv[]);`
- `int execvp(const char *file, char *const argv[]);`

## Question

- Rank the following calls in order of safety (high to low):
  - system
  - execl
  - execv
  - execvp
  
- a) execv, execvp, execl, system
- b) execl, execv, execvp, system
- c) execvp, execl, execv, system

## Question

- Rank the following calls in order of safety (high to low):
  - system
  - execl
  - execv
  - execvp
- a) execv, execvp, execl, system
- b) execl, execv, execvp, system**
- c) execvp, execl, execv, system

# File Descriptors

- UNIX: forked and exec'ed processes inherit file descriptors
- Remember the principle of complete mediation
- Close all unneeded file descriptors (before calling exec)

# Questions?

---

## About These Slides

- You are free to copy, distribute, display, and perform the work; and to make derivative works, under the following conditions.
  - You must give the original author and other contributors credit
  - The work will be used for personal or non-commercial educational uses only, and not for commercial activities and purposes
  - For any reuse or distribution, you must make clear to others the terms of use for this work
  - Derivative works must retain and be subject to the same conditions, and contain a note identifying the new contributor(s) and date of modification
  - For other uses please contact the Purdue Office of Technology Commercialization.
- Developed thanks to the support of Symantec Corporation

# **Pascal Meunier** **pmeunier@purdue.edu**

Contributors:

Jared Robinson, Alan Krassowski, Craig Ozancin, Tim Brown, Wes Higaki, Melissa Dark, Chris Clifton, Gustavo Rodriguez-Rivera

