



Systems and Internet
Infrastructure Security

Network and Security Research Center
Department of Computer Science and Engineering
Pennsylvania State University, University Park PA

Advanced Systems Security: Confused Deputy

Trent Jaeger

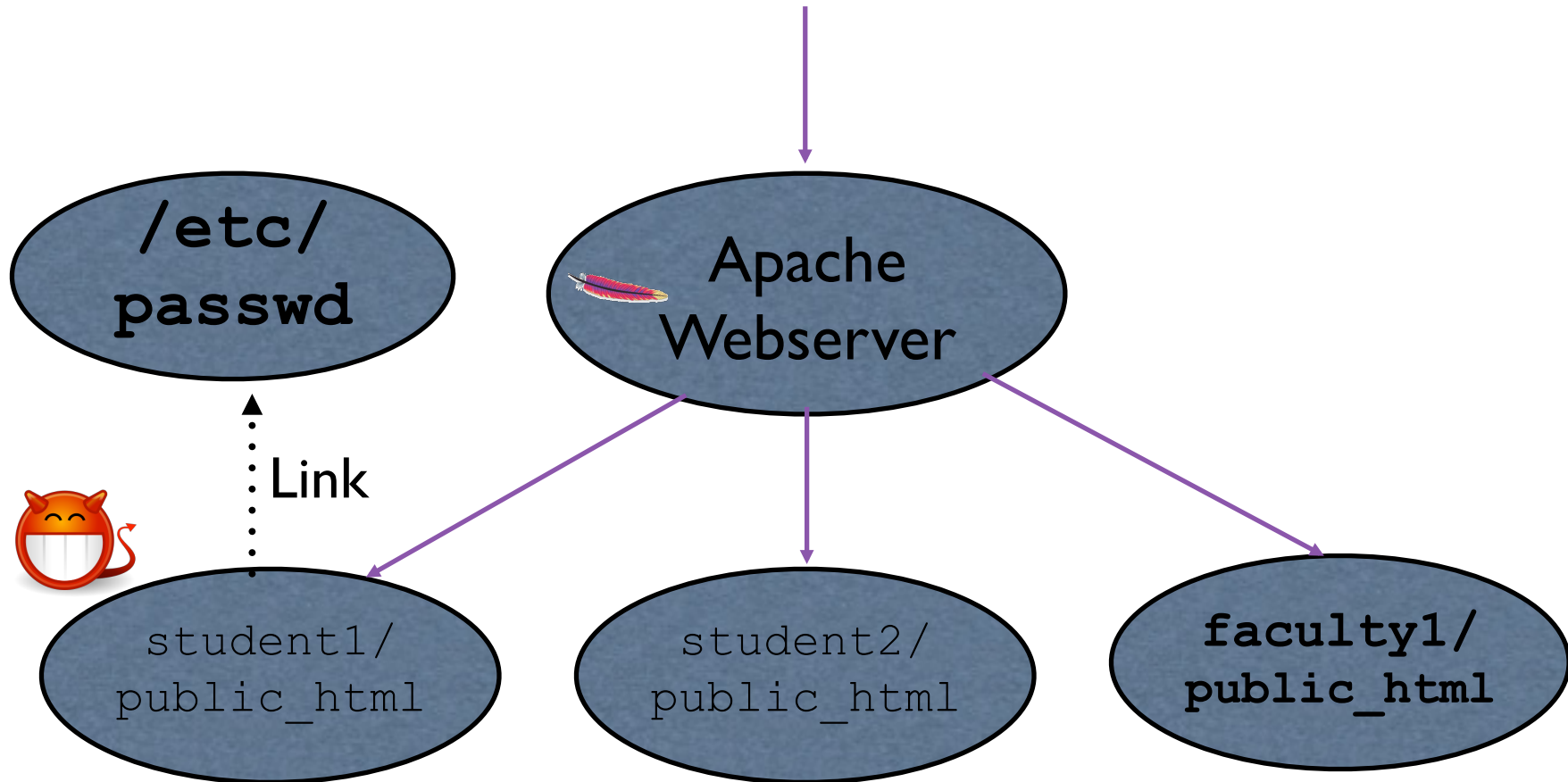
*Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University*

Talk Outline

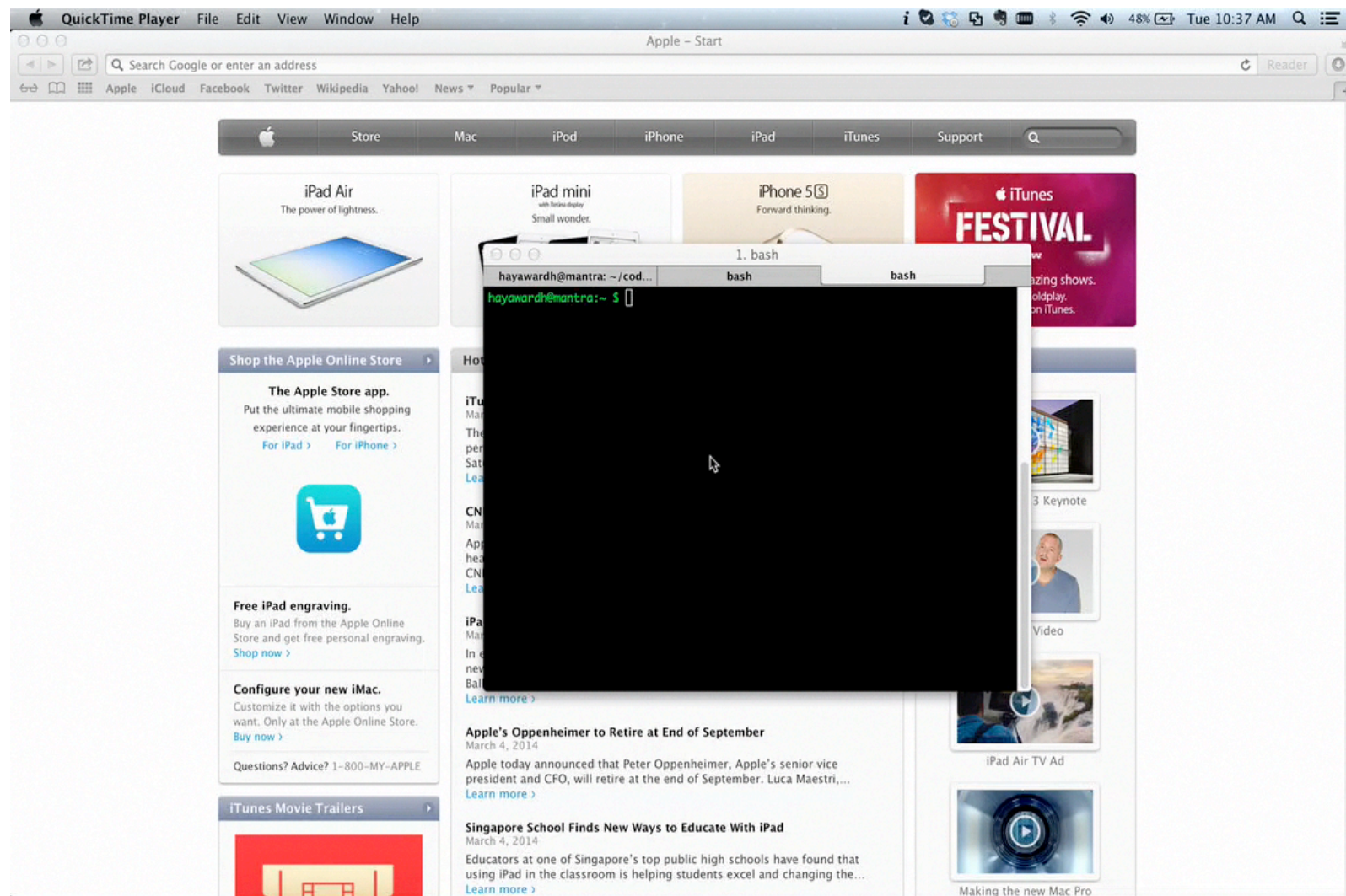
- **Problem:** Processes need resources from system
 - Just a simple `open(filename, ...)` right?
 - But, adversaries can redirect victims to resources of their choosing

A Webserver's Story ...

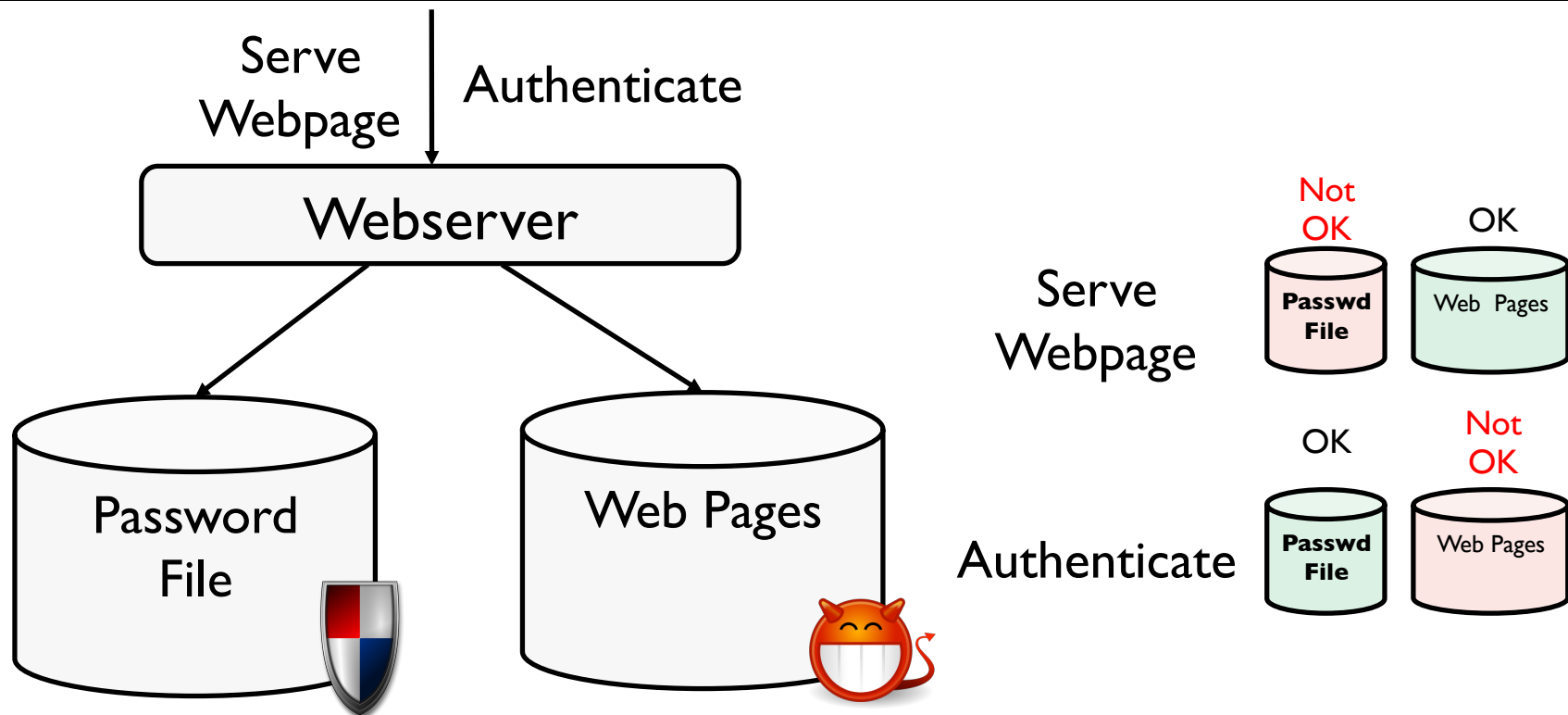
- Consider a university department webserver ...
GET /~student1/index.html HTTP/1.1







Attack Video



What Just Happened?

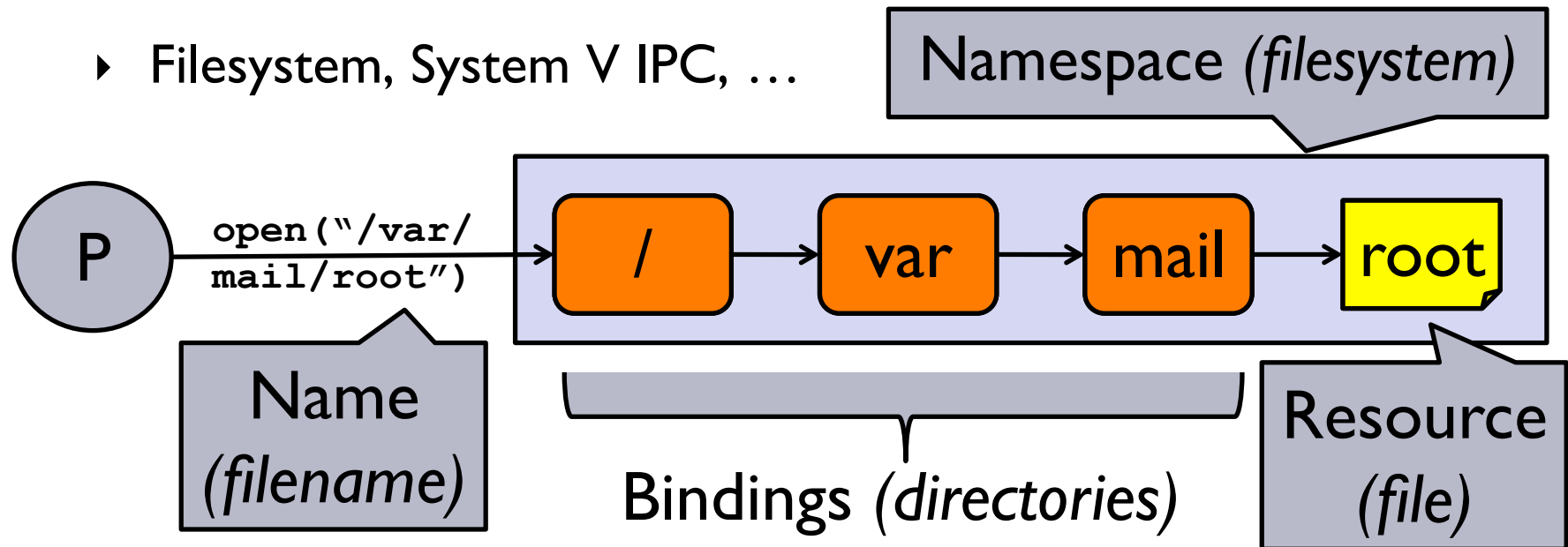


- Program acts as a *confused deputy*

- ▶  when expecting 
- ▶  when expecting 

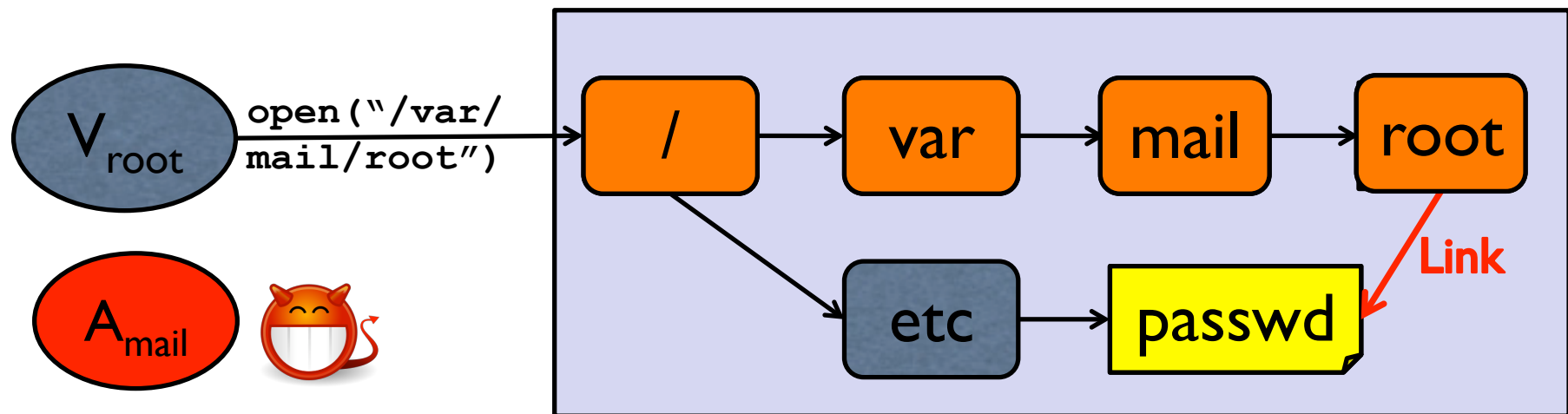
Name Resolution

- Processes often use *names* to obtain access to *system resources*
- A *nameserver* (e.g., OS) performs *name resolution* using *namespace bindings* (e.g., *directory*) to convert a *name* (e.g., *filename*) into a system *resource* (e.g., *file*)
 - Filesystem, System V IPC, ...



Link Traversal Attack

- Adversary controls **bindings** to direct a victim to a resource not normally accessible to the adversary
- Victim **expects** adversary-accessible resource, gets a protected resource instead
 - May take advantage of race conditions (**TOCTTOU attacks**)

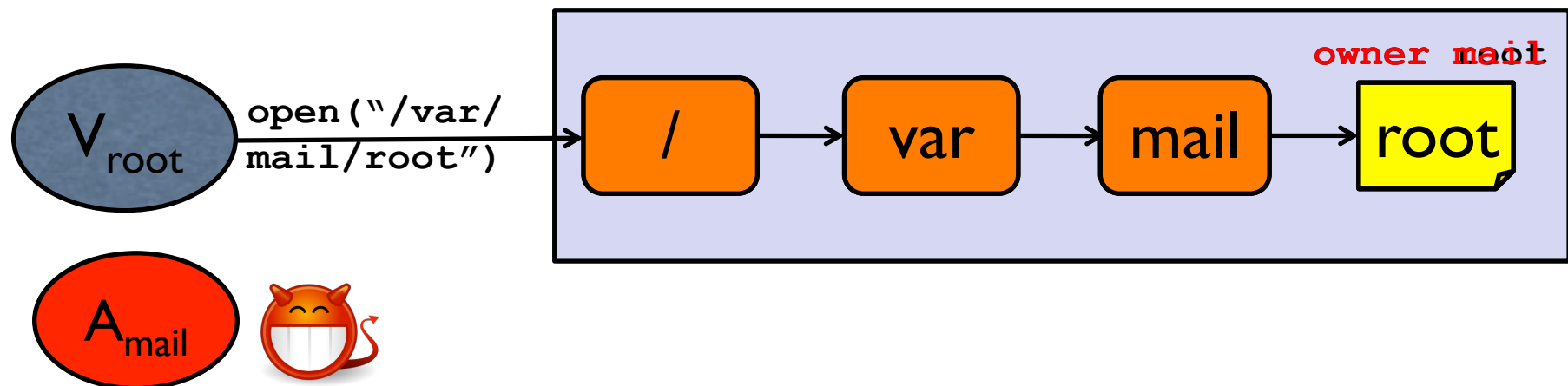


TOCTTOU Attacks

- Time-of-check-to-time-of-use Attack
- Check System Call
 - ▶ Does the requesting party have access to the file? (stat, access)
 - ▶ Is the file accessed via a symbolic link? (lstat)
- Use System Call
 - ▶ Convert the file name to a file descriptor (open)
 - ▶ Modify the file metadata (chown, chmod)

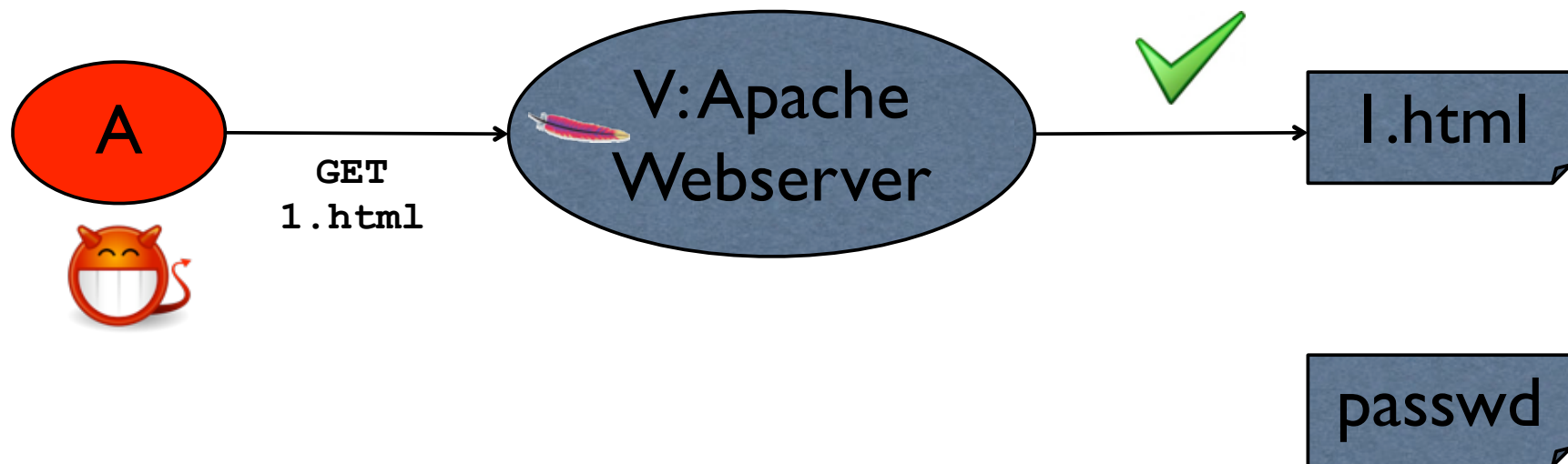
File Squatting Attack

- Adversary controls final **resource** enabling the adversary to control input that the victim may depend on
- Victim **expects** protected resource, gets an adversary-controlled resource instead



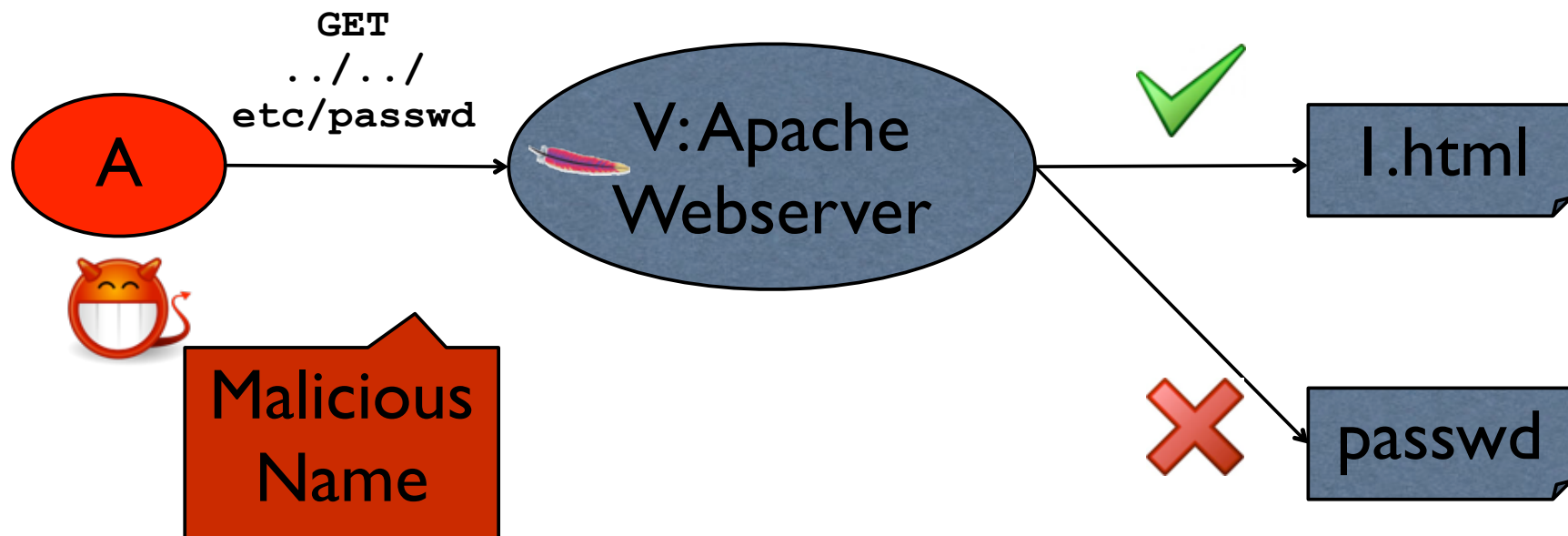
Directory Traversal

- Adversary controls the **name** to direct victim to an adversary inaccessible (high integrity) resource



Directory Traversal

- Adversary controls the **name** to direct victim to an adversary inaccessible (high integrity) resource
- Victim **expects** adversary accessible (low integrity) resource



Confused Deputy Attacks

PHP File Inclusion
CWE-98

TOCTTOU Races
CWE-362

Follow

Search
ath

**Confused
Deputy
Attacks**

File
squattin
CWE-205

Directory Traversal
CWE-22

Untrusted Library
Load
CWE-426

Prevalence

Attack Class	CWE class	CVE Count	
		<2007	2007-12
Untrusted Search Path	CWE-426	109	329
Untrusted Library Load	CWE-426	97	91
File/IPC squat	CWE-283	13	9
Directory Traversal	CWE-22	1057	1514
PHP File Inclusion	CWE-98	1112	1020
Link Following	CWE-59	480	357
TOCTTOU Races	CWE-362	17	14
Signal Races	CWE-479	9	1
% Total CVEs	-	12.40%	9.41%

Integrity (and Secrecy) Threat

- **Confused Deputy**

- ▶ *Process is tricked into performing an operation on an adversary's behalf that the adversary could not perform on their own*
 - Write to (read from) a privileged file



Attacks Easily Overlooked

- Manual checks can easily overlook vulnerabilities
- Misses file squat at line 03!

```
01 /* filename = /var/mail/root */
02 /* First, check if file already exists */
03 fd = open (filename, flg);
04 if (fd == -1) {
05     /* Create the file */
06     fd = open(filename, O_CREAT|O_EXCL);
07     if (fd < 0) {
08         return errno;
09     }
10 }
11 /* We now have a file. Make sure
12 we did not open a symlink. */
13 struct stat fdbuf, filebuf;
14 if (fstat (fd, &fdbuf) == -1)
15     return errno;
16 if (lstat (filename, &filebuf) == -1)
17     return errno;
18 /* Now check if file and fd reference the same file,
19    file only has one link, file is plain file. */
20 if ((fdbuf.st_dev != filebuf.st_dev
21     || fdbuf.st_ino != filebuf.st_ino
22     || fdbuf.st_nlink != 1
23     || filebuf.st_nlink != 1
24     || (fdbuf.st_mode & S_IFMT) != S_IFREG)) {
25     error (_("%s must be a plain file
26         with one link"), filename);
27     close (fd);
28     return EINVAL;
29 }
30 /* If we get here, all checks passed.
31    Start using the file */
32 read(fd, ...)
```

Squat during
create (resource)

Symbolic link

Hard link,
race conditions

Mandatory Access Control



- Does MAC solve this problem?
 - What does SELinux say?

Prior Work - Defenses

- TOCTTOU Attack known since 1973 at least
- Proven impractical to produce system-only defenses
- Track file metadata
 - Leverage **extended POSIX API** (fstat, lstat) to track name resolution
 - Cowan, Dean-Hu, Tsafrir, ...
- Track system calls
 - Maintain a **table of past system calls** to detect when an unexpected resource is retrieved
 - Tsyrklevich-Yee, Calvin Ko, ...
- **All were shown to be flawed**

Prior Work - Defenses

- Cai et al 2009 demonstrated that system-only defenses
 - ▶ “all kernel-based dynamic race detectors must have a model of the programs they protect or provide imperfect protection.”
- Consider the “atom race” defenses
 - ▶ Calls lstat(2), access(2), open(2), fstat(2) for k rounds
 - ▶ Can be circumvented by – sLaAsOF (LaAsOFC)^k
 - Where *a* represents the attacker’s action of switching the atom to point to an accessible file, and *s* represents the act of switching atom to point to the secret file

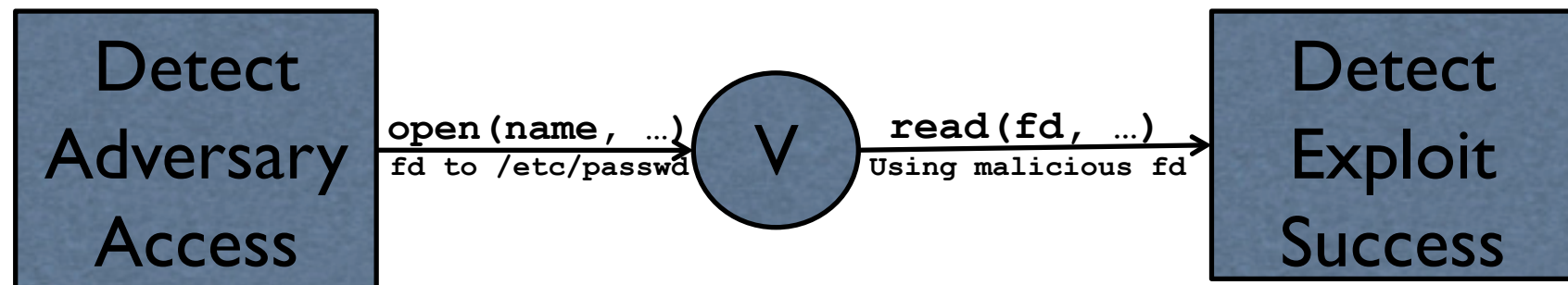
Runtime Analysis

- Run program and detect system call sequences that may be vulnerable
- Still, many **false positives**
 - Program code might defend itself
 - And may be inaccessible to adversaries
 - In our study, “only” 13% of adversary-accessible name resolutions are vulnerable
- **False negatives**
 - Attacks require special conditions
 - Current working directory, links, ...

```
01 /* filename = /var/mail/root */
02 /* First, check if file already exists */
03 fd = open (filename, flg);
04 if (fd == -1) {
05     /* Create the file */
06     fd = open(filename, O_CREAT|O_EXCL);
07     if (fd < 0) {
08         return errno;
09     }
10 }
11 /* We now have a file. Make sure
12 we did not open a symlink. */
13 struct stat fdbuf, filebuf;
14 if (fstat (fd, &fdbuf) == -1)
15     return errno;
16 if (lstat (filename, &filebuf) == -1)
17     return errno;
18 /* Now check if file and fd reference the same file,
19 file only has one link, file is plain file. */
20 if ((fdbuf.st_dev != filebuf.st_dev
21     || fdbuf.st_ino != filebuf.st_ino
22     || fdbuf.st_nlink != 1
23     || filebuf.st_nlink != 1
24     || (fdbuf.st_mode & S_IFMT) != S_IFREG)) {
25     error (_("%s must be a plain file
26         with one link"), filename);
27     close (fd);
28     return EINVAL;
29 }
30 /* If we get here, all checks passed.
31 Start using the file */
32 read(fd, ...)
```


STING [USENIX 2012]

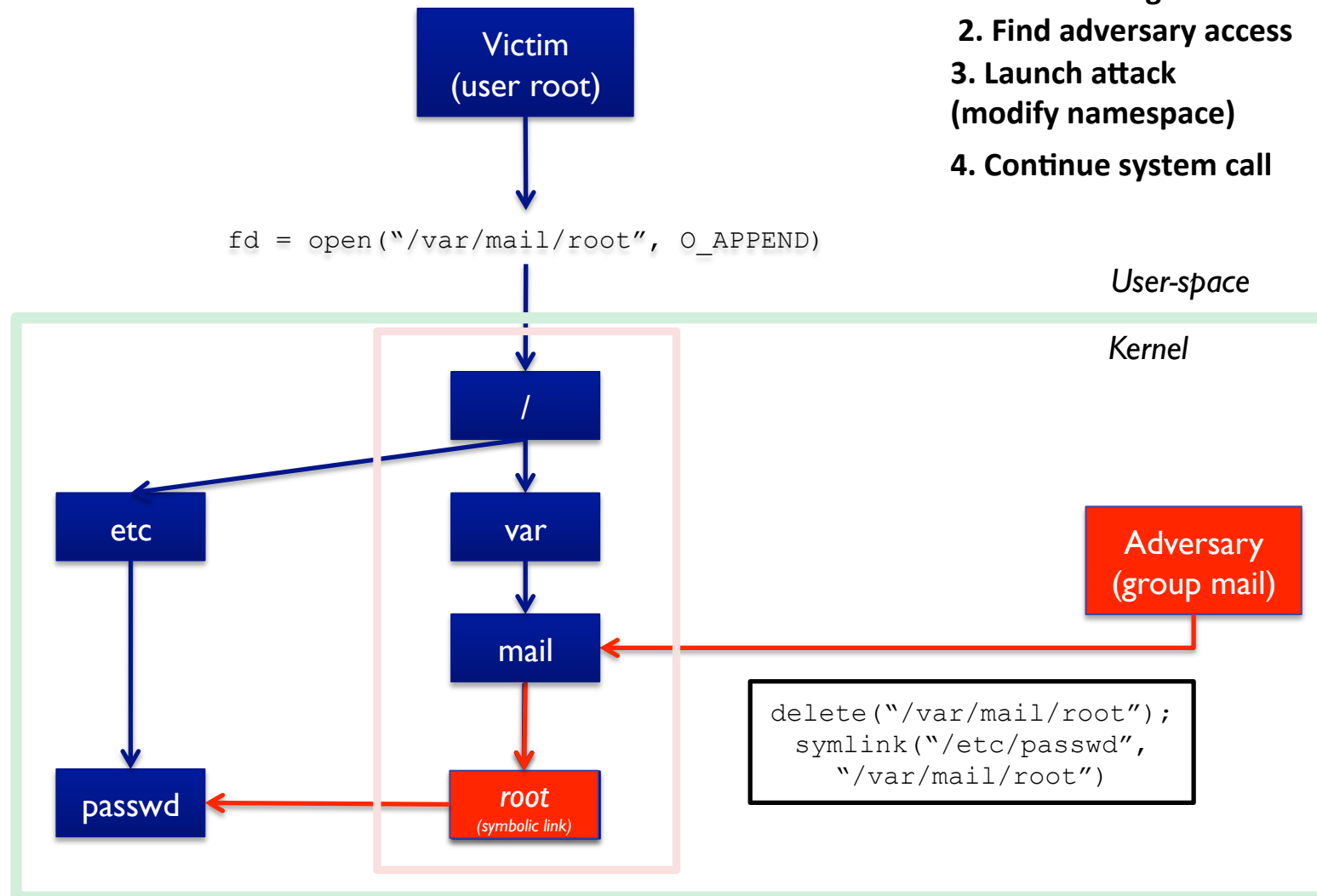
- We **actively change** the namespace whenever an adversary can write to a binding used in resolution
 - ▶ **Fundamental problem:** **adversaries may be able to write directories used in name resolution**
- Use adversary model to identify program adversaries and vulnerable directories [ASIACCS 2012]



Vulnerable!

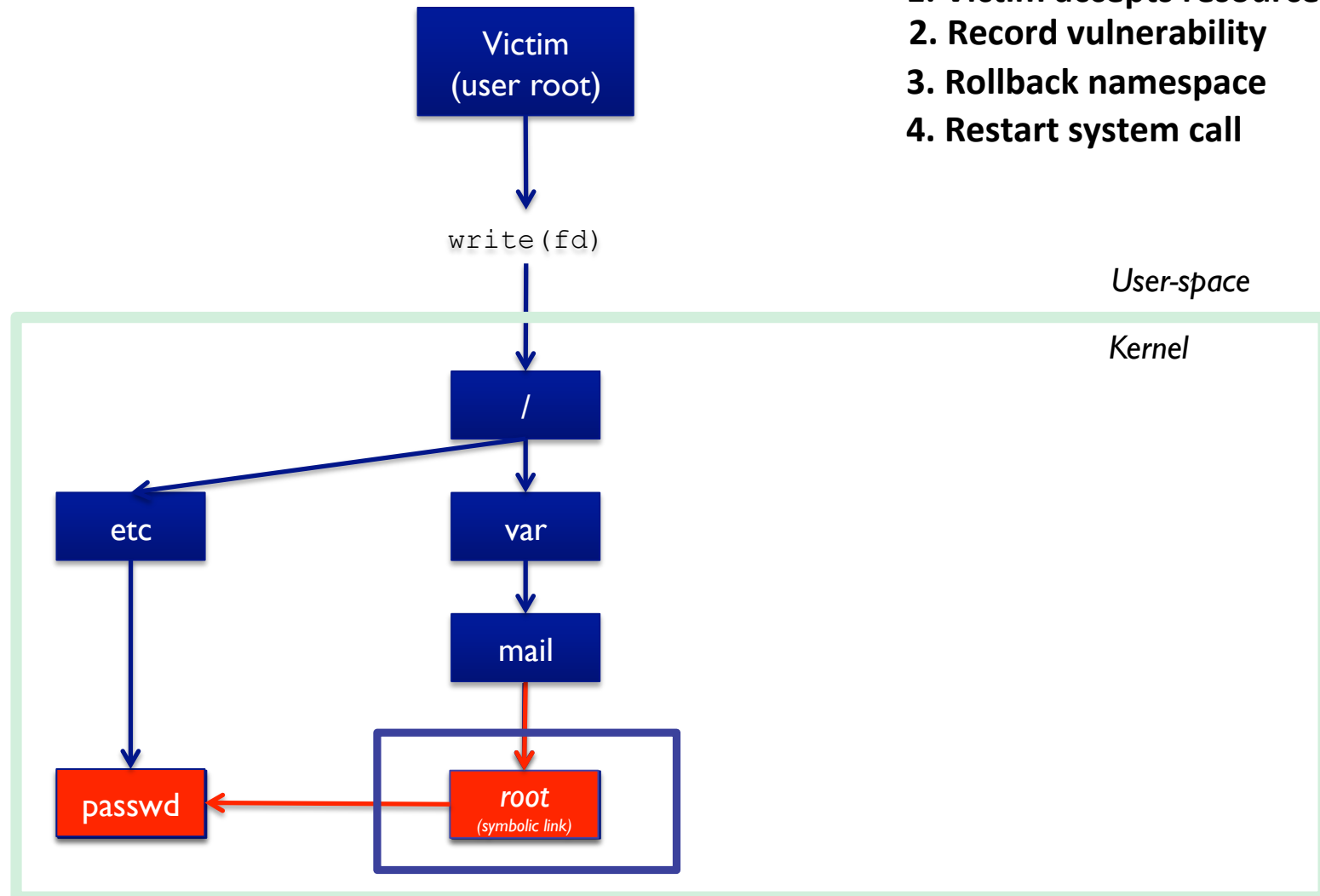
STING Launch Phase

1. Find bindings
2. Find adversary access
3. Launch attack (modify namespace)
4. Continue system call



STING Detect Phase

1. Victim accepts resource
2. Record vulnerability
3. Rollback namespace
4. Restart system call



STING Detects TOCTTOU Races

- STING can **deterministically create races**, as it is in the OS

Victim

Adversary

```
SOCKET_DIR=/tmp/.X11-unix

set_up_socket_dir () {
  if [ "$VERBOSE" != no ]; then
    log_begin_msg "Setting up $SOCKET_DIR..."
  fi
  if [ -e $SOCKET_DIR ] && [ ! -d $SOCKET_DIR ]; then
    mv $SOCKET_DIR $SOCKET_DIR.$$
  fi
  mkdir -p $SOCKET_DIR
  chown root:root $SOCKET_DIR
  chmod 1777 $SOCKET_DIR
  do_restorecon $SOCKET_DIR
  [ "$VERBOSE" != no ] && log_end_msg 0 || return 0
}
```

```
ln -s /etc/passwd
/tmp/.X11-unix
```

Results - Vulnerabilities

Program	Vuln. Entry	Priv. Escalation DAC: uid->uid	Distribution	Previously known
dbus-daemon	2	messagebus->root	Ubuntu	Unknown
landscape	4	landscape->root	Ubuntu	Unknown
Startup scripts (3)	4	various->root	Ubuntu	Unknown
mysql	2	mysql->root	Ubuntu	1 Known
mysql_upgrade	1	mysql->root	Ubuntu	Unknown
tomcat script	2	tomcat6->root	Ubuntu	Known
lightdm	1	*->root	Ubuntu	Unknown
bluetooth-applet	1	*->user	Ubuntu	Unknown
java (openjdk)	1	*->user	Both	Known
zeitgeist-daemon	1	*->user	Both	Unknown
mountall	1	*->root	Ubuntu	Unknown
mailutils	1	mail->root	Ubuntu	Unknown
bsd-mailx	1	mail->root	Fedora	Unknown
cupsd	1	cups->root	Fedora	Known
abrt-server	1	abrt->root	Fedora	Unknown
yum	1	sync->root	Fedora	Unknown
x2gostartagent	1	*->user	Extra	Unknown
19 Programs	26			21 Unknown

Both old
and new
programs
Special
users to
root
Known
but
unfixed!

Program Defense

- Check for symbolic link (lstat)
- Check for lstat-open race
- Check for inode recycling
- Do checks for each path component (**safe_open**)
 - ▶ /, var, mail, ...
- Cai et al found that **races can be won > 50% of time**
 - ▶ E.g., long sequence of symlinks

```
/* fail if file is a symbolic link */
int open_no_symlink(char *fname)
{
01  struct stat lbuf, buf;
02  int fd = 0;
03  lstat(fname, &lbuf);
04  if (S_ISLNK(lbuf.st_mode))
05    error("File is a symbolic link!");
06  fd = open(fname);
07  fstat(fd, &buf);
08  if ((buf.st_dev != lbuf.st_dev) ||
09      (buf.st_ino != lbuf.st_ino))
10    error("Race detected!");
11  lstat(fname, &lbuf);
12  if ((buf.st_dev != lbuf.st_dev) ||
13      (buf.st_ino != lbuf.st_ino))
14    error("Cryogenic sleep race!");
15  return fd;
}
```

Safe Open - Inefficient

- Checking retrieved resources is expensive
 - ▶ Single open() requires $4 * \text{path length}$ additional syscalls
 - ▶ Programmers omit checks to improve performance
- Example: **Apache documentation recommends switching off resource access checks**

FollowSymLinks and SymLinksIfOwnerMatch

Wherever in your URL-space you do not have an `Options FollowSymLinks`, or you do have an `Options SymLinksIfOwnerMatch` Apache will have to issue extra system calls to check up on symlinks. One extra call per filename component. For example, if you had:

```
DocumentRoot /www/htdocs
<Directory />
    Options SymLinksIfOwnerMatch
</Directory>
```

and a request is made for the URI `/index.html`. Then Apache will perform `lstat(2)` on `/www`, `/www/htdocs`, and `/www/htdocs/index.html`. The results of these `lstats` are never cached, so they will occur on every single request. If you really desire the symlinks security checking you can do something like this:

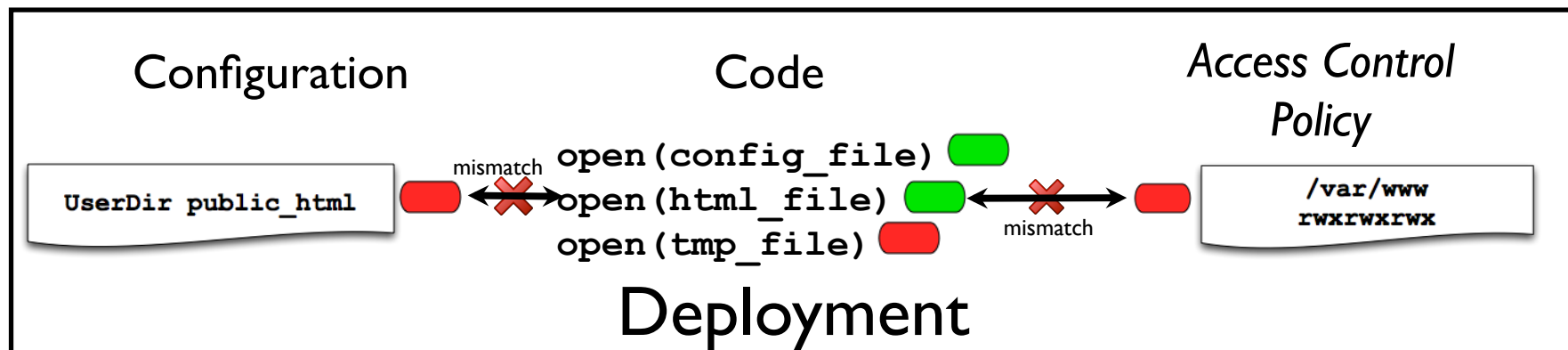
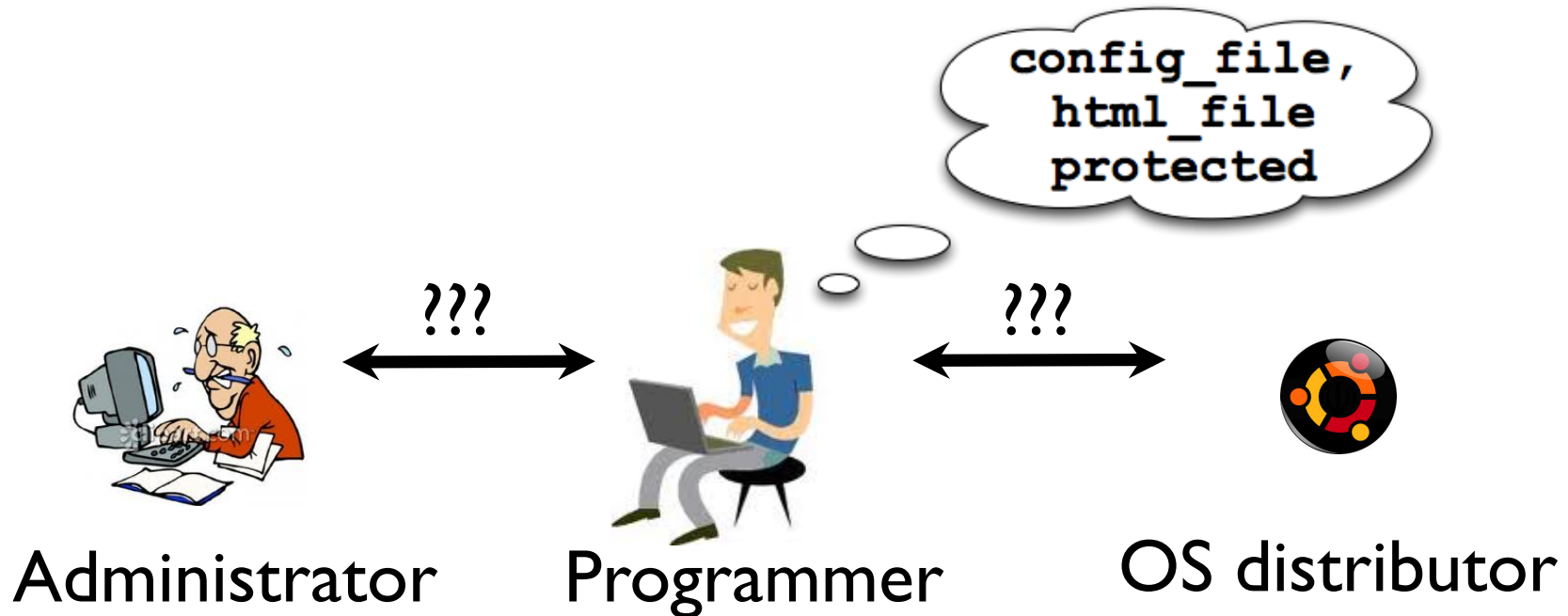
```
DocumentRoot /www/htdocs
<Directory />
    Options FollowSymLinks
</Directory>

<Directory /www/htdocs>
    Options -FollowSymLinks +SymLinksIfOwnerMatch
</Directory>
```





This at least avoids the extra checks for the `DocumentRoot` path. Note that you'll need to add similar sections if you have any `Alias` or `RewriteRule` paths outside of your document root. **For highest performance, and no symlink protection, set `FollowSymLinks` everywhere, and never set `SymLinksIfOwnerMatch`.**

Cause - Multiple Parties

Expectations mismatch, blame each other



Solution Overview

- Match programmer expectation onto system
 - ▶ Irrespective of OS access control or admin configuration
 - ▶ If programmer expects to access only  , then they should not access 
 - Unexpected attack surface
 - ▶ If programmer expects  , then they should not access 
 - Classic confused deputy

Solution Overview

- $\{P\}$ - System calls where programmer expects adversary control
- $\{S\}$ - System calls in deployment that adversaries actually control
- $\{R\}$ - System calls in deployment that retrieve adversary-accessible resources
- When programmer expects no adversary control, block adversary-controlled system calls
 - ▶ Prevent unexpected adversary control: $S \subseteq P$
- When adversary control happens, limit adversary to accessible resources:
 - ▶ Prevent confused deputy: *for all x , if x in $S \rightarrow x$ in R*

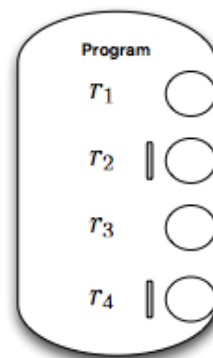
To Find Mismatches

- Need a model that describes
 - How a program performs resource access.
 - How do programs build names, bindings?
 - What are programmer expectations for resource access?
 - If they expect adversary access to names, bindings: *protect*
 - If not: *do nothing* 😊



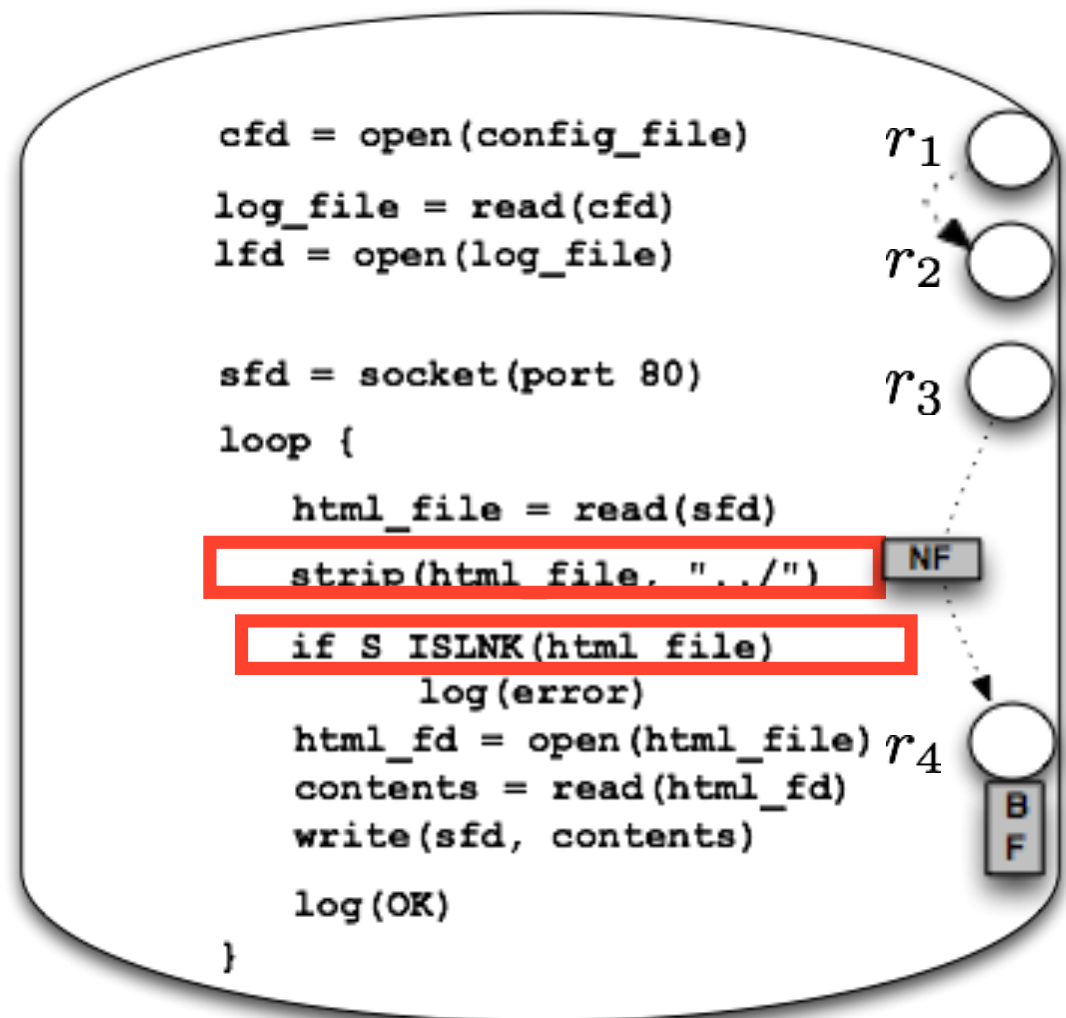
Programmer Expectations

- Can we determine where a programmer expects adversarial control of resource access?
- Strawman solution
 - Ask programmers to add annotations in code
- Insight: There are already annotations (sort of) --
 - *Filters (defensive code)!*




Resource Access Filters

- Write defensive checks (filters) to protect resource accesses
 - Name filters
 - Binding filters



Filters as Annotations

- *Heuristic*: If programmer expects adversarial control of resource access, she will add name/binding filters
 - ▶ *Corollary*: No filter \Rightarrow access only 

Determine P from filters


- No filter \implies not in expected attack surface P
- If no binding filter

Base Case 1  $\implies r_1 \notin P$

- If no name filter on an outgoing name flow

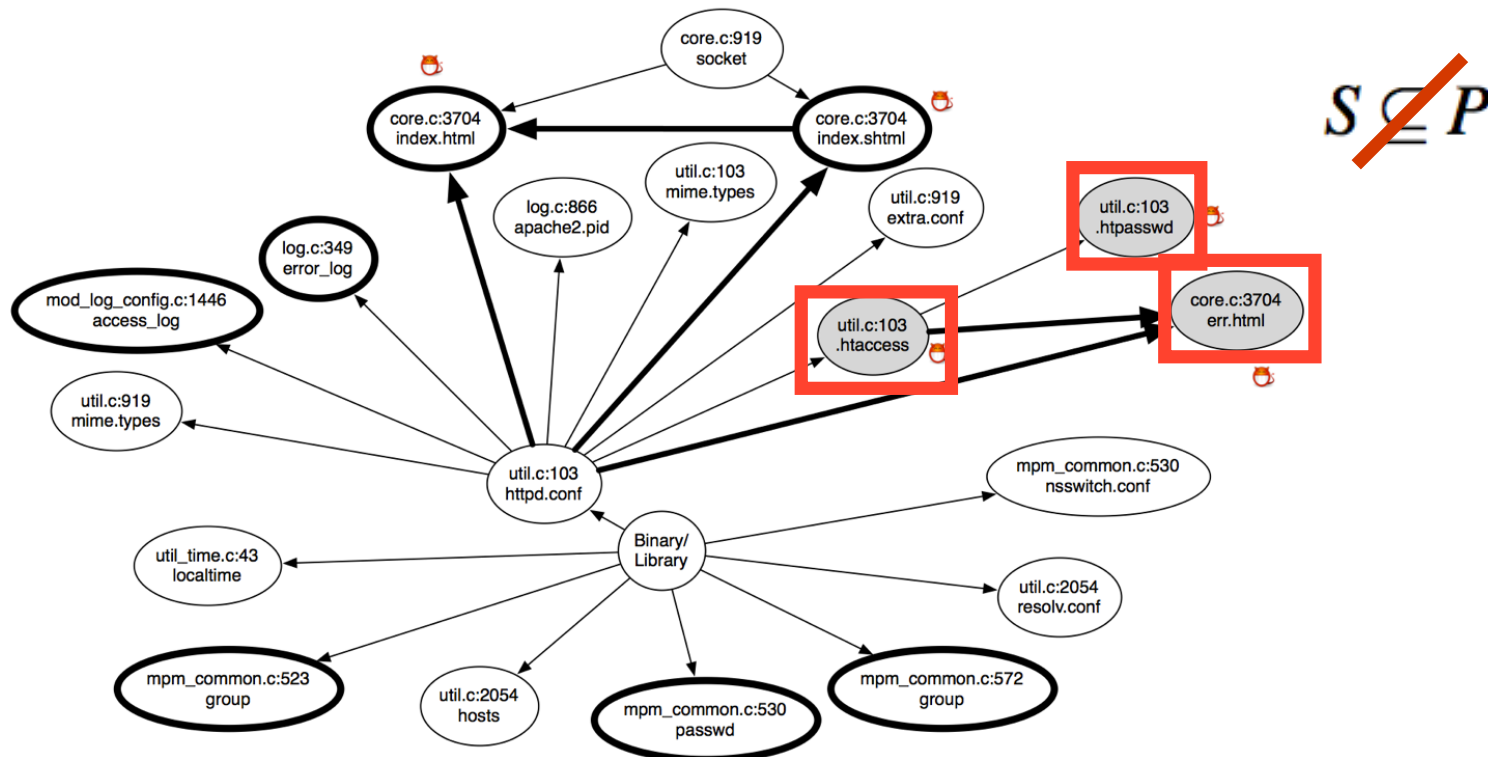
Base Case 2  $\implies r_1 \notin P$

- If a resource access not in P is reachable

Transitive Closure  $\implies r_1 \notin P$

- Any remaining resource accesses are in P

Runtime Mismatches



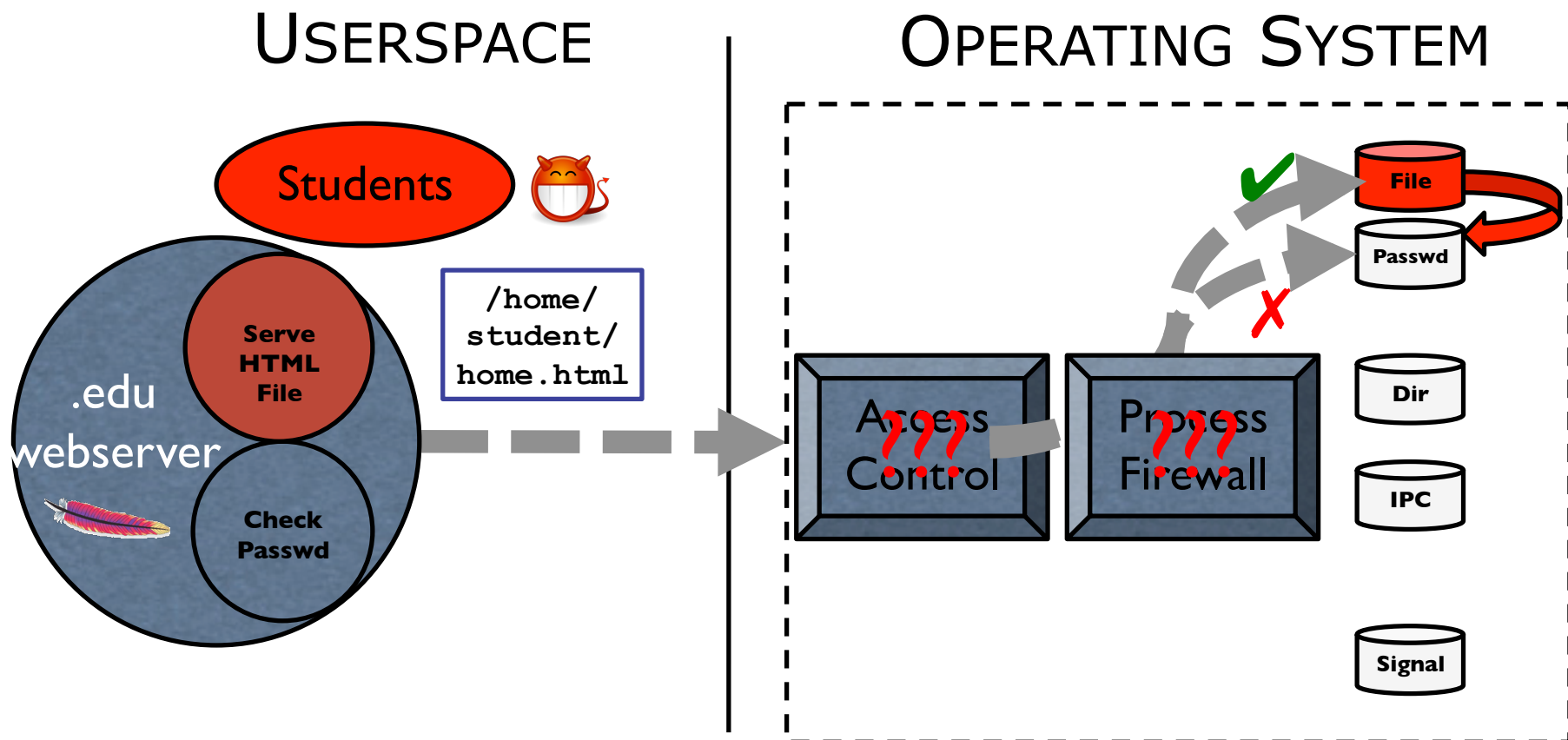
Design Goals

Develop a system defense that blocks processes from using permissions that lead to exploit

- Should **not require programmer** code changes
- Should be **capable of protecting processes** with resource access vulnerabilities
- Should be **efficient (faster than in program)**
- Should be possible to configure policies **automatically** with **no false positives**

No Program Change

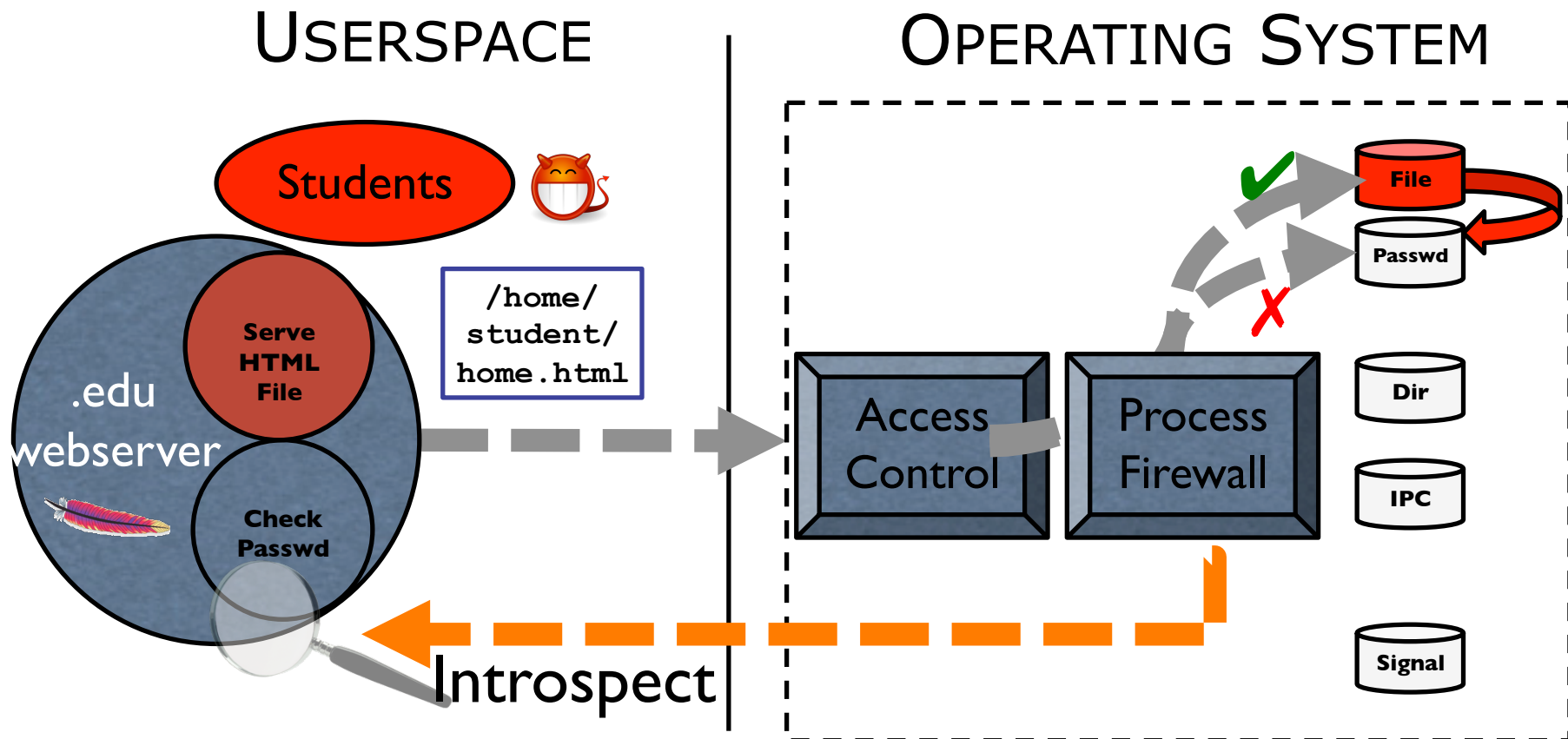
How do we block attacks without changing program?



System defense that blocks unsafe resources

Identify System Call

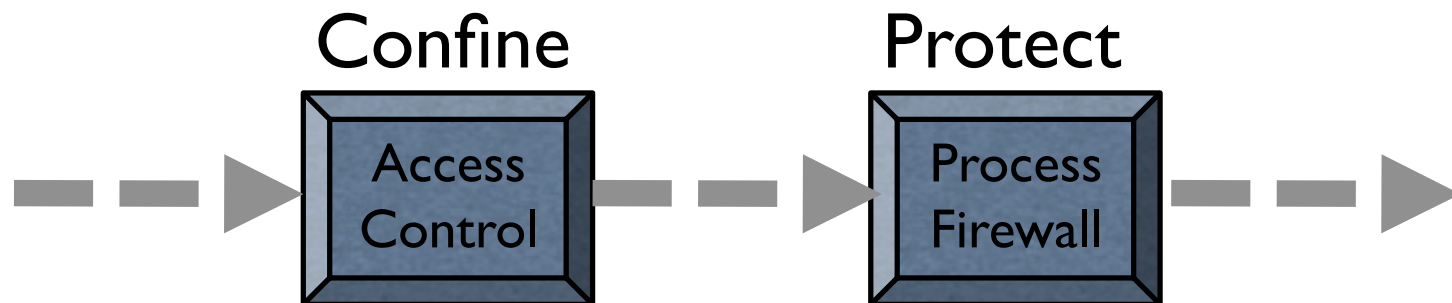
How do we distinguish different system calls?



Process Context: Entrypoint, Call Stack, etc.

Process Introspection

- Why can we introspect into the process?
 - What about mimicry attacks (on IDS)?
- The Process Firewall *protects* victim processes instead of *confining* adversary processes
 - Mimicry only invalidates process's own protection
 - Depend on access control for confinement

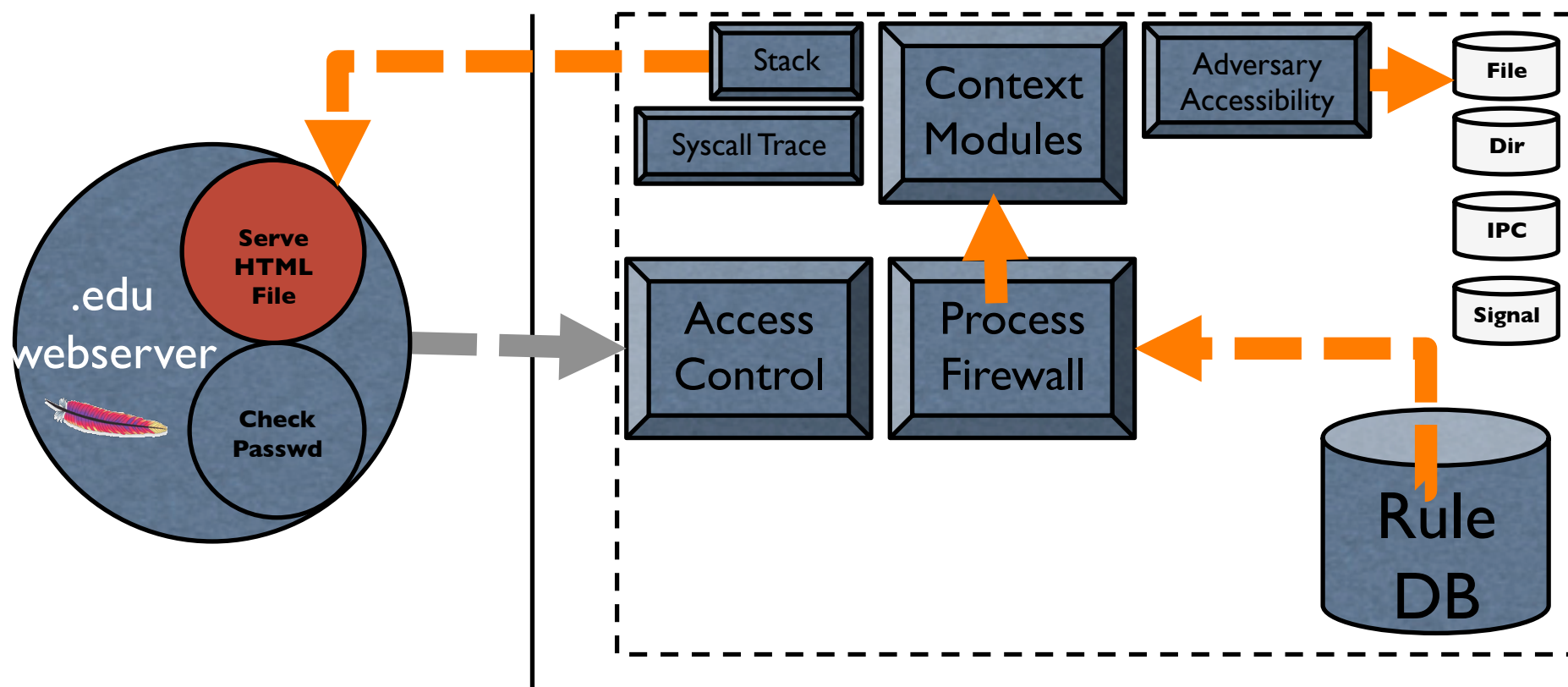


Gathering Context

Extensibly Gather Context?

USERSPACE

OPERATING SYSTEM



Context modules gather process context and resource properties required to evaluate rules

Performance Overhead

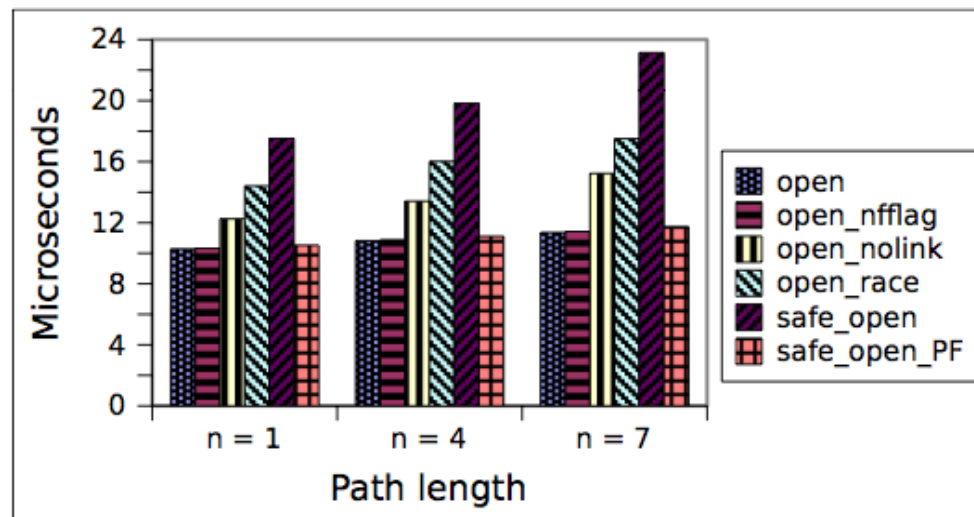
- Macrobenchmarks showed under 2-4% overhead (with 500 rules)

Benchmark	Mean \pm 95% CI (% overhead)			
	Without PF	PF Base		PF Full
Apache Build (s)	73.67 \pm 0.06	72.82 \pm 0.12	(0.2)	75.61 \pm 0.06 (4.0)
Boot (s)	14.49 \pm 0.10	14.51 \pm 0.2	(0.0)	14.82 \pm 0.1 (2.2)
Web1-L (ms)	0.946 \pm 0.001	0.947 \pm 0.001	(0.1)	0.967 \pm 0.002 (2.2)
Web1-T (Kb/s)	467.67 \pm 0.1	465.45 \pm 0.3	(0.5)	455.35 \pm 0.8 (2.5)
Web1000-L (ms)	0.963 \pm 0.002	0.967 \pm 0.005	(0.4)	0.992 \pm 0.012 (3.0)
Web1000-T (Kb/s)	459.15 \pm 0.1	455.14 \pm 0.7	(0.9)	444.04 \pm 1.2 (3.2)

Faster Than Program?

Should resource access checks be in program code?

- We measured performance of `safe_open()` in program against equivalent Process Firewall rules
 - 103% in program vs 2.3% as Process Firewall rules



Process Firewall rules much more efficient!

Evaluation - Expectation

<i>Program</i>	<i>Dev Tests?</i>	$ V $	$ E $	$ V_f $	$ E_f $	$\in P$	$\notin P$	<i>Impl. Exp. %</i>	<i>Missing</i>	<i>Redundant</i>	<i>Vulns.</i>	<i>Inv. 1s</i>	<i>Inv. 2s</i>
Apache v2.2.22	Yes*	20	23	7	5	7	13	65%	2	0	3	13	12
OpenSSH v5.3p1	Yes	17	17	14	0	14	3	17.6%	0	3	0	3	2
Samba3 v3.4.7	Yes	210	84	78	19	78	132	62.8%	0	5	0	132	40
Winbind v3.4.7	Yes	50	38	19	13	19	31	63.3%	0	0	0	31	28
Postfix v2.10.0	No	181	15	79	7	79	102	56.32%	0	9	0	102	15

- In 4/5 programs, programmers implicitly expect **> 55%** of resource accesses to never be adversary controlled in **any** deployment
 - OpenSSH most secure
- We found 2 missing checks that corresponded to 2 **previously-unknown** vulnerabilities and 1 default misconfiguration in the Apache webserver

.htpasswd Vulnerability

- Apache allows users to specify a password file to protect their pages in .htaccess

```
AuthUserFile /home/userh/.htpasswd
AuthType Basic
AuthName "My Files"
Require valid-user
```

- Neither name flow nor binding is filtered
 - User can specify **any** password file, even of other users, or the system-wide /etc/passwd (if in proper format)
- Can be used to brute-force passwords
 - No rate limit on HTTP auth (unlike terminal logins)
- Vulnerability hidden all these years, showing importance of automated and principled reasoning of resource access

Vulnerability

- Typical example of resource access vulnerability
 - ▶ Who is to blame?
 - Admin for not recognizing adversaries and improper configuration?
 - OS distributor for default insecure configuration?
 - Programmer for providing the configuration option?
- Difficult to tell, but the name flow enforcement can block vulnerability without requiring code or access control policy change

