Synchronization and State

Synchronization and State

Adrian Coleșa

Universitatea Tehnică din Cluj-Napoca
Computer Science Department

November 23, 2015

The purpose of this lecture

1. Presents basic synchronization aspects
2. Presents common vulnerabilities related to improper synchronization
3. Describes good practices to write code safe to synchronization attacks

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Definition and context

- concurrent executions (threads, processes)
- shared resources
- race conditions $\Rightarrow$
  - inconsistent resources' state
  - unpredicted and undesired results
- race conditions could be
  1. *inherent* to the application logic
  2. triggered by (unexpected) *external* events

Example of Non-reentrant (unsynchronized) Function

```
struct list *global_list;
int global_list_count;
int list_add(struct list *element)
{
  struct list *tmp;
  if(global_list_count > MAX_ENTRIES)
    return -1;

  for(list = global_list; list->next; list = list->next);

  list->next = element;
  element->next = NULL;
  global_list_count++;
  return 0;
}
```

Example of a Function Vulnerable to External Interference

```
if (file_not_accessible("user_config_file", user_id))
    return -1;

int fd = open("user_config_file", O_RDWR);
display_file(fd);
```

Security Implications

- race conditions occur in security-critical code, such as
  - recording whether a user is authenticated
  - manipulating important state information that should not be influenced by an outsider
- *untrusted* interfering code sequence
  - can be authored directly by the attacker
  - typically it is external to the vulnerable program
- could be vulnerable even if there is no direct concurrency inside an application
  - race conditions (concurrency) not expected

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Possible Consequences

- availability
  - bypass resource cleanup routines or trigger multiple initialization routines $\Rightarrow$ resource exhaustion (DoS)
  - lead the program into unexpected states $\Rightarrow$ crash
- confidentiality and integrity
  - race condition combined with predictable resource names and loose permissions $\Rightarrow$ overwrite or access confidential data

Protection Needed

1. using synchronization mechanisms, i.e. *establish internal access rules*
2. deny or eliminate the possibilities of external interferences, i.e. protect against attacks

Affected Languages

- C / C++
- Java
- language independent

CWE References

- CWE-362: Race Condition (`https://cwe.mitre.org/data/definitions/362.html`)
- CWE-364: Signal Handler Race Condition
- CWE-365: Race Condition in Switch
- CWE-366: Race Conditions Within a Thread
- CWE-367: Time-of-Check to Time-of-Use (TOCTOU) Race Condition
- CWE-368: Context Switching Race Condition
- CWE-370: Race Condition in Checking for Certificate Revocation

CWE References (cont.)

- CWE-421: Race Condition During Access to Alternate Channel
- `http://www.cvedetails.com/cwe-details/362/Race-Condition.html`

Real Life Examples

- see all at `http://www.cvedetails.com/vulnerability-list/cweid-362/vulnerabilities.html`
- see `https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&cwe_id=CWE-362`
- CVE-2004-1235 (Linux kernel)
  - race condition in the (1) `load_elf_library` and (2) `binfmt_aout` function calls for uselib in Linux kernel 2.4 allows local users to execute arbitrary code by manipulating the VMA (virtual memory areas) descriptors

Real Life Examples (cont.)

- description and PoC `http://www.isec.pl/vulnerabilities/isec-0021-uselib.txt`
- CVE-2015-7820
  - race condition in the administration-panel web service in IBM System Networking Switch Center (SNSC) before 7.3.1.5 and Lenovo Switch Center before 8.1.2.0 allows remote attackers to obtain privileged-account access, and consequently provide ZipDownload.jsp input containing directory traversal sequences to read arbitrary files, via a request to port 40080 or 40443
- CVE-2015-5754

Real Life Examples (cont.)

- race condition in runner in Install.framework in the Install Framework Legacy component in Apple OS X before 10.10.5 allows attackers to execute arbitrary code in a privileged context via a crafted app that leverages incorrect privilege dropping associated with a locking error
- CVE-2015-2706
  - race condition in the AsyncPaintWaitEvent::AsyncPaintWaitEvent function in Mozilla Firefox before 37.0.2 allows remote attackers to execute arbitrary code or cause a denial of service (use-after-free) via a crafted plugin that does not properly complete initialization
- CVE-2015-1882

Real Life Examples (cont.)

- multiple race conditions in IBM WebSphere Application Server (WAS) 8.5 Liberty Profile before 8.5.5.5 allow remote authenticated users to gain privileges by leveraging thread conflicts that result in Java code execution outside the context of the configured EJB Run-as user
- CVE-2014-9710
  - the Btrfs implementation in the Linux kernel before 3.19 does not ensure that the visible xattr state is consistent with a requested replacement, which allows local users to bypass intended ACL settings and gain privileges via standard filesystem operations (1) during an xattr-replacement time window, related to a race condition

Real Life Examples (cont.)

or (2) after an xattr-replacement attempt that fails because the data does not fit

Detection Methods

- black box testing
- white box testing
- automated dynamic analysis
- automated static analysis
- manual code review
- formal methods

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Overview

- steps
  1. check the state of a resource before using it
  2. use the resource if state is good
- problem: resource's state changed between check and use
- vulnerability: attacker change the resource's state to take some advantage
- see Matt Bishop, Michael Dilger, "*Checking for Race Conditions in File Accesses*", 1996

Overview (cont.)

- existence of such an interval: *programming condition*
- *programming interval*: the interval itself
- *environmental condition*: the attacker be able to affect the assumptions created by the program's first action
- $\Rightarrow$ **both conditions must hold for an exploitable TOCTTOU binding flaw**

TOCTOU Vulnerable Examples

```
void main(int argc, char **argv)
{
    int fd;
    if (access(argv[1], W_OK) != 0)
        exit(1);
    fd = open(argv[1], O_RDWR);
    /* Use fd... */
}
```

- context: a privileged (SUID) application checks if real UID has access to a file
- file could be changed between `access()` and `open()`
- called TOCTOU binding flaw

TOCTOU Vulnerable Examples (cont.)

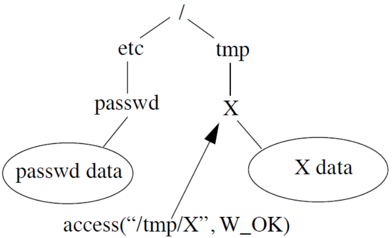

Figure 1a.

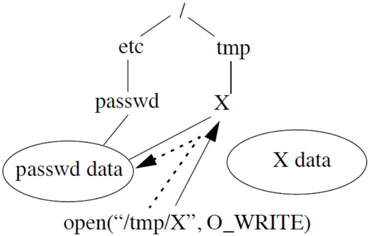

Figure 1b.

TOCTOU Vulnerable Examples (cont.)

```
void deltree(char *dir)
{
    chdir(dir);
    /* Recursively delete
    contents of dir ... */
    chdir("..");
}
```

- move *dir* while the programming is traversing the sub-tree beneath *dir*,
- $\Rightarrow$ cause the program to delete files it did not intend to delete

TOCTOU Vulnerable Examples (cont.)

```
int mktmpfile(char *fname)
{
    int fd = -1;
    struct stat buf;
    if (stat(fname, &buf) < 0)
        fd = open(fname, O_CREAT, S_IRWXU);
    return fd;
}
```

- create the file before the victim does
- ⇒ control the permissions and owner of the file
- ⇒ cause the program to open some other file that already exists on the system

TOCTOU Vulnerable Examples (cont.)

```
int run(char *exe)
{
    struct stat s[3];
    lstat(exe, &s[0]);
    stat(exe, &s[1]);
    if (s[0].st_uid != s[1].st_uid)
        exit(1);
    lstat(exe, &s[2]);
    setreuid(s[2].st_uid, s[2].st_uid);
    execl(exe, NULL);
}
```

- modifies the symbolic link *exe* either immediately before or after the last call to `lstat`
- $\Rightarrow$ can execute arbitrary code as another user

TOCTOU Other Examples

root	attacker
	mkdir("/tmp/etc")
	creat("/tmp/etc/passwd")
readdir("/tmp")	
lstat("/tmp/etc")	
readdir("/tmp/etc")	
	rename("/tmp/etc","/tmp/x")
	symlink("/etc","/tmp/etc")
unlink("/tmp/etc/passwd")	

(a) garbage collector

root	attacker
lstat("/mail/ann")	
	unlink("/mail/ann")
	symlink("/mail/ann","/etc/passwd")
fd = open("/mail/ann")	
write(fd,...)	

(b) mail server

root	attacker
access(filename)	
	unlink(filename)
	link(sensitive,filename)
fd = open(filename)	
read(fd,...)	

(c) setuid

Symlinks and Cryogenic Sleep

```
if (lstat(fname, &stb1) >= 0 && S_ISREG(stb1.st_mode)) {
    fd = open(fname, O_RDWR);
    if (fd < 0 || fstat(fd, &stb2) < 0
        || ino_or_dev_mismatch(&stb1, &stb2))
        raise_big_stink();
} else {
    /* do the O_EXCL thing */
}
```

- context: reopen files in "`/tmp`"
- attack
  1. create the expected regular file in "/tmp"
  2. stop application (sending it SIGSTOP) between `lstat()` and `open()`
  3. record the device and inode number of the regular file, remove it, and ...

Symlinks and Cryogenic Sleep (cont.)

4. **wait** (possibly very long) until another file with the same values is created
5. resume application (by sending it SIGCONT)
6. there could be techniques to increase the chance

Outline

Outline

UNIX System V Process Synchronization

- provides semaphores
- sets of semaphores
- operations
  - create / get access to a semaphore set

    ```
    int semget(key_t key, int sem_no, int flags);
    ```

  - atomic execution of multiple operations on the semaphore set

    ```
    int semop(int sem_id, struct sembuf *ops, unsingned op_no);
    ```

  - control the semaphore set

    ```
    int semctl(int sem_id, int sem_no, int cmd, ...);
    ```

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Windows Process Synchronization

- mechanisms to synchronize threads of a process or processes in the system
- synchronization objects
  - types: mutexes, events, semaphores, waitable timers
  - states: signaled and unsignaled
- could be named or unnamed
- share the same namespace with jobs and file-mappings

Windows Wait Functions

- wait (a max time) for a synchronization object to become signaled

```
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
```

- wait (a max time) for one or more synchronization objects to become signaled

```
DWORD WaitForMultipleObjects(DWORD count, const HANDLE *lpHandles,
                             BOOL bWaitAll, DWORD dwMilliseconds);
```

Windows Mutex Objects

- creating a (named or unnamed) mutex

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpMutexAttributes,
                   BOOL bInitialOwner, LPCSTR lpName);
```

- if the mutex exists, it is opened, ignoring the other parameters
- opening an existing mutex

```
HANDLE OpenMutex(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCSTR lpName);
```

- signaling (releasing) a mutex

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Windows Mutex Objects (cont.)

- signaling requirements
    - mutex ownership
    - `MUTEX_MODIFY_STATE` access right
- could be taken recursively, but must be released the same no of times

Windows Event Objects

- wake-up more threads simultaneously (broadcast the event)
- manual-reset type
  - stays in the signaled state until manually set to non-signaled
- auto-reset type
  - automatically reset to the non-signaled state after waking-up (signaling) a thread
- creating the event object

```
HANDLE CreateEvent(LPSECURITY_ATTRIBUTES lpEventAttributes
                   BOOL bManualReset, BOOL bInitialState,
                   LPCSTR lpName);
```

Windows Event Objects (cont.)

- opening an existing event object

```
HANDLE OpenEvent(DWORD dwDesiredAccess, BOOL bInheritHandle
                 LPCSTR lpName);
```

- setting an event object to signaled state

```
BOOL SetEvent(HANDLE hEvent);
```

- resting an manual-reset event object to unsignaled state

```
BOOL ResetEvent(HANDLE hEvent);
```

- set/reset requires `EVENT_MODIFY_STATE` access right

Windows Semaphore Objects

- creating a semaphore

```
HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
                       LONG lInitialCount, LONG lMaximumCount,
                       LPCSTR lpName);
```

- opening an existing semaphore

```
HANDLE OpenSemaphore(DWORD dwDesiredAccess, BOOL bInheritHandle,
                     LPCSTR lpName);
```

- releasing /incrementing a semaphore

```
BOOL ReleaseSemaphore(HANDLE hSemaphore, LONG lReleaseCount,
                      LPLONG lpPreviousCount);
```

Windows Waitable Timer Objects

- used for scheduling tasks to be executed by a thread after a time interval
- types: manual-reset and synchronization
- could be periodic
- creating a new timer

```
HANDLE CreateWaitableTimer(LPSECURITY_ATTRIBUTES lpTimerAttributes
                           BOOL bManualReset, LPCSTR lpName);
```

- opening an existing timer

```
HANDLE OpenWaitableTimer(DWORD dwDesiredAccess, BOOL bInheritHandle,
                         LPCSTR lpName);
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Windows Waitable Timer Objects (cont.)

- initializing a timer

```
BOOL SetWaitableTimer(HANDLE hTimer, const LARGE_INTEGER *pDueTime,
                      LONG lPeriod, PTIMERAPCROUTINE pfnCompletionRoutine,
                      LPVOID lpArgToCompletionRoutine,
                      BOOL fResume);
```

- canceling a timer

```
BOOL CancelWaitableTimer(HANDLE hTimer);
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Lack of Use

- missing using synchronization objects when needed could lead to unexpected results
- example

```
char *users[NUSRES];
int crt_idx = 0;

DWORD phoneConferenceThread(SOCKET s)
{
  char *name;

  name = readString(s);
  if ((NULL == name) || (crt_idx >= NUSERS))
    return 0;

  users[crt_idx] = name;
  crt_idx++;
  ...
}
```

Lack of Use (cont.)

- could lead to user array corruption
  - overwrite a (privileged) user with another (non-privileged) one
  - overflow the array

Lack of Use: Vulnerable Example

- https://defuse.ca/race-conditions-in-web-applications.htm

```
function withdraw($amount)
{
   $balance = getBalance();
   if($amount <= $balance)
   {
       $balance = $balance - $amount;
       echo "You have withdrawn: $amount";
       setBalance($balance);
   }
   else
   {
       echo "Insufficient funds.";
   }
}
```

Lack of Use: Vulnerable Example (cont.)

Thread 1	Thread 2

($10)
`function withdraw($amount)`
`{` ($10,000)
`$balance = getBalance();`
`if($amount <= $balance)`
`{` ($9,990)
`$balance = $balance - $amount;`
`echo "You have withdrawn: $amount";`	

	($10)
	`function withdraw($amount)`
	`{` ($10,000)
	`$balance = getBalance();`
	`if($amount <= $balance)`
	`{` ($9,990)
	`$balance = $balance - $amount;`
	`echo "You have withdrawn: $amount";`
	`setBalance($balance);` ($9,990)
	`}`
	`else`
	`{`
	`echo "Insufficient funds.";`
	`}`
	`}`

`setBalance($balance);` ($9,990)
`}`
`else`
`{`
`echo "Insufficient funds.";`
`}`
`}`	

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Incorrect Use of Synchronization Objects

- application specific
- could lead to data corruption and/or deadlock, even without an attacker interference
- the attacker could try to create the race condition context to gain advantage from
- variant: do not check the return value (success or not) of the synchronization functions

Squatting With Named Synchronization Objects

- context
  - creation of a new synchronization object
  - a synchronization object with the same name cold already exist
- case 1: do not check for new object creation success
  - the attacker creates before the application an object with the same name
  - $\Rightarrow$ could take ownership of the synchronization object
  - change the synchronization objects (e.g. take locks, change semaphores values, signal events etc.)
  - $\Rightarrow$ control /corrupt the application execution

Squatting With Named Synchronization Objects (cont.)

- example 1 (Windows)

```
hMutex = CreateMutex(MUTEX_MODIFY_STATE, TRUE, "MyMutex");
if (NULL == hMutex)
    return -1;
...
ReleaseMutex(hMutex);
```

- example 2 (Linux)

```
int semid = semget(ftok("/home/user/file", 'A'), 10, IPC_CREATE | 0600);
...
```

- case 2: check for new object creation success
  - attacker could cause *denial of service*
  - example 1 (Windows)

Squatting With Named Synchronization Objects (cont.)

```
hMutex = CreateMutex(MUTEX_MODIFY_STATE, TRUE, "MyMutex");
if ((NULL == hMutex) ||
    (GetLastError() == ERROR_ALREADY_EXISTS))
  return FALSE;
```

- example 2 (Linux)

```
int semid = semget(ftok("/home/user/file", 'A'), 10,
                   IPC_CREATE | IPC_EXCL | 0600);
if (semid < 0)
  return -1;
...
```

- case 3: create the object with too much permissions
  - attacker could change the synchronization object
  - example (Linux)

Squatting With Named Synchronization Objects (cont.)

```
int semid = semget(IPC_PRIVATE, 10, IPC_CREATE | 0666);
if (semid < 0)
  return -1;
...
```

Code Review

1. synchronization object scoreboards
   - object name
   - object type
   - using purpose
   - instantiated
   - instantiation parameters
   - permissions
   - used by
   - notes
2. lock matching
   - check for execution paths not releasing a lock
   - limitations: applicable only for locks

Outline

Outline

Mutexes (Locks)

- initialization

```
pthread_mutex lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex lock = PTHREAD_RECURSIVE_INTIALIZER_NP; // Linux
pthread_mutex lock = PTHREAD_ERRORCHECK_MUTEX_NP; // Linux
int pthread_mutex_init(pthread_mutex_t *mutex, const
                pthread_mutex_attr_t *attr)
```

- lock a mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- unlock a mutex

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- remove a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Condition Variables

- initialization

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
```

- waiting on a condition variable

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                          const struct timespec *abstime);
```

- signal a condition variable

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- remove a condition variable

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Outline

Critical Sections

- declared in code as `CRITICAL_SECTION` data type
- visible only to threads of the same process
- initialization

```
void InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);

BOOL InitializeCriticalSectionAndSpinCount(
                                LPCRITICAL_SECTION lpCriticalSection,
                                DWORD dwSpinCount);
```

- entering (exclusively, i.e. acquire) in a critical section

```
void EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Critical Sections (cont.)

- exiting (i.e. release) the critical section

```
void LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

- remove a critical section associated structures

```
void DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Outline

Code Review: Strategy to Avoid Race Conditions

1. identify shared resources that are acted on by multiple threads
2. determine whether the appropriate locking mechanism has been selected
3. examine the code that modifies shared resource
   - check if locking mechanisms have been neglected or misused

Ensure the Appropriate Locking Mechanisms Are Used

- determine the resource access requirements
- check if chosen synchronization mechanisms meet the requirements, like
  - mutual exclusion $\Rightarrow$ lock, critical section
  - more threads can read simultaneously $\Rightarrow$ semaphore

Examine Accesses to the Object

- problem 1: no synchronization mechanism used
- problem 2: incorrect usage of synchronization mechanisms
- problem 3: miss release a synchronization mechanism an some paths
- problem 4: do not check for return values of the synchronization API
- problem 5: deadlock and starvation

Incorrect Usage Example

```
struct element *queue;
pthread_mutex_t queue_lock;
pthread_cond_t queue_cond;
int fd;

void *job_task(void *arg)
{
    struct element *elem;

    pthread_mutex_init(&queue_lock, NULL);

    for(;;)
    {
        pthread_mutex_lock(&queue_lock);

        if(queue == NULL)
            pthread_cond_wait(&queue_cond, &queue_lock);

        elem = queue;
        queue = queue->next;

        pthread_mutex_unlock(&queue_lock);
```

Incorrect Usage Example (cont.)

```
        .. process element ..
    }
    return NULL;
}
void *network_task(void *arg)
{
    struct element *elem, *tmp;
    struct request *req;

    pthread_mutex_init(&queue_lock, NULL);

    for(;;)
    {
        req = read_request(fd);

        if(req == NULL) // bad request
            continue;

        elem = request_to_job_element(req);
        free(req);

        if(elem == NULL)
            continue;
```

Incorrect Usage Example (cont.)

```
        pthread_mutex_lock(&queue_lock);

        if(queue == NULL)
        {
            queue = elem;
            pthread_cond_broadcast(&queue_cond);
        }
        else
        {
            for(tmp = queue; tmp->next; tmp = tmp->next);
            tmp->next = elem;
        }

        pthread_mutex_unlock(&queue_lock);
    }
}
```

No Result Checked Example

```
DWORD processJob(LPVOID arg)
{
    struct element *elem;

    for(;;)
    {
        WaitForSingleObject(hMutex, MAX_TIME);

        if(queue == NULL)
            WaitForSingleObject(queueEvent, MAX_TIME);

        elem = queue;
        queue = queue->next;
        ReleaseMutex(hMutex);

        .. process element ..
    }
    return 0;
}
```

Deadlock Example

```
struct interface *interfaces[MAX_INTERFACES];

int packet_process(int num)
{
    struct interface *in = interfaces[num];
    struct packet *pkt;

    for(;;)
    {
        pthread_mutex_lock(in->lock);
        pthread_cond_wait(in->cond_arrived, in->lock);
        pkt = dequeue_packet(in);

        if(needs_forwarding(pkt))
        {
            int destnum;
            struct interface *dest;

            destnum = find_dest_interface(pkt);
            dest = interfaces[destnum];

            pthread_mutex_lock(dest->lock);
```

Deadlock Example (cont.)

```
            enqueue_packet(pkt, dest);
            pthread_mutex_unlock(dest->lock);

            in->stats[FORWARDED]++;

            pthread_mutex_unlock(in->lock);

            continue;
        }
        pthread_mutex_unlock(in->lock);
        .. process packet ..
    }
}
```

Outline

Outline

Description

- UNIX specific software interrupts raised by the kernel
- signal handling
  - ignore (excepting SIGKILL and SIGSTOP)
  - block, setting a signal mask (excepting SIGKILL and SIGSTOP)
  - handle asynchronously
  - default action (usually terminate the process)

Sending Signals

- done using the `kill()` system call
- requires permissions
  - Linux: the root or senders having real UID /saved SUID equal to receiver's real UID / saved SUID
  - BSD: the root or senders having real / effective equal to receiver's real / effective UID
- BSD case: a privileged process assuming the privileges of a normal user exposed to signals from that user
- Linux (obsolete) strategy: FS UID (a different UID used just for FS accesses)

Signal Handling

- using the `signal()` function

```
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

  - explicit handler
  - `SIG_IGN`
  - `SIG_DFL`

Signal Handling (cont.)

- using the `sigaction()` function

```
struct sigaction {
  void (sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t*, void*);
  sigset_t sa_mask;
  int sa_flags;
};

int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oact);
```

  - the handler alternatives are exclusive
  - some signals could be masked while handling the specified signal

Signal Handling (cont.)

- masking/unmasking signals with the `sigprocmask()` function

```
int sigprocmask(int how, const sigset_t *set, sigset_t oset);
```

  - `SIG_BLOCK`
  - `SIG_UNBLOCK`
  - `SIG_SETMASK`

Jump Locations

- context: return to a point in a program from any other point in certain conditions
- functions: `setjmp()`, `longjmp()`

```
int setjmp(jmp_buf env);
int longjmp(jmp_buf env, int val);
```

- could be used by signal handlers
- `setjmp()` used to set a return (jump) point used by the `longjmp()`
  - save the context (e.g. IP, stack) to be restored
  - a return 0 indicate a direct call (setting the point)
  - a non-0 return indicate a jump

Jump Locations (cont.)

- `longjump` does normally not return
- example

```
jmp_buf env;

int process_msg(int sock)
{
  for (;;) {
    if (setjmp(env) != 0)
      log("Invalid␣request.␣Ignore␣msg.");

    if (read_packet_header(sock) < 0)
      return -1;

    switch(header.type) {
      case USER:
        parse_username_request(socket);
        break;
      case PASS:
        parse_password_request(socket);
        break;
```

Jump Locations (cont.)

```
      case OPEN:
        parse_openfile_request(socket);
        break;
      ...
    }
  }
}

int open_file_validation(char *filename)
{
  if (strstr(filename, ".."))
    longjmp(env);
  ...
}
```

- functions: `sigsetjmp(), siglongjmp()`

```
int sigsetjmp(sigjmp_buf env, int savesigs);
int siglongjmp(sigjmp_buf env, int val);
```

Outline

Asynchronous Safe Functions

- program execution could be interrupted any moment by an non-masked signal
- if signal handled, the current execution is suspended and resumed
- asynchronous-safe function
  - run correctly even if interrupted by an asynchronous event
  - $\Rightarrow$ should be re-entrant
  - $\Rightarrow$ should deal correctly with signal interruptions
- signal handlers should also be asynchronous-safe

Basic Interruption

- handler relies on some sort of global program state
  - e.g. global variables expected to be initialized, when they are not
- example of vulnerable code

```
char *user = NULL;

int cleanup(int sig)
{
  printf("caught signal: cleaning up ...\n");
  free(user);
  exit(1);
}

int main()
{
  char buf[1024];
  signal(SIGINT, cleanup);
  ...
```

Basic Interruption (cont.)

```
    ... read from a file into buffer ...
    user = malloc(strlen(buf) + 1);
    strncpy(user, buffer, strlen(buf));
    ...
    free(user);
    ...
}
```

- problems
  - trying release a NULL pointer
  - multiple releases of *user*
  - if signal handler not exits and *strncpy* would be interrupted $\Rightarrow$ access a released memory
- general form of vulnerable exiting signal handler

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Basic Interruption (cont.)

```
int sig_handler(int signo)
{
  ...
  cleanup_function(...);
  exit(0);
}

some_other_function(...)
{
  ...
  cleanup_function(...);
}
```

- the `cleanup_function` must be re-entrant
- all the function called by `cleanup_function` must be re-entrant

Non-Returning Signal Handlers

- does not return execution control back to the interrupted function
  - explicitly terminate the process by calling `exit()`
  - return to another part of the application using `longjmp()`
- `longjmp()` aspects
  - it is generally safe for a `longjmp()` to simply terminate the program
  - any code reachable via the signal handler by using `longjmp()` must be asynchronous-safe
- example: Sendmail SMTP server signal race vulnerability

Non-Returning Signal Handlers (cont.)

```
void
collect(fp, smtpmode, hdrp, e, rsetsize)
    SM_FILE_T *fp;
    bool smtpmode;
    HDR **hdrp;
    register ENVELOPE *e;
    bool rsetsize;
{
    ... other declarations ...
    volatile time_t dbto;
    ...
    dbto = smtpmode ? TimeOuts.to_datablock : 0;

    /*
    ** Read the message.
    **
    ** This is done using two interleaved state machines.
    ** The input state machine is looking for things like
    ** hidden dots; the message state machine is handling
    ** the larger picture (e.g., header versus body).
    */

    if (dbto != 0)
```

Non-Returning Signal Handlers (cont.)

```
{
    /* handle possible input timeout */
    if (setjmp(CtxCollectTimeout) != 0)
    {
        if (LogLevel > 2)
            sm_syslog(LOG_NOTICE, e->e_id,
                    "timeout waiting for input from %s
                    during message collect",
                    CURHOSTNAME);
        errno = 0;
        if (smtpmode)
        {
            /* Override e_message in usrerr() as this
            ** is the reason for failure that should
            ** be logged for undelivered recipients.
            */
            e->e_message = NULL;
        }
        usrerr("451 4.4.1 timeout waiting for input
            during message collect");
        goto readerr;
    }
    CollectTimeout = sm_setevent(dbto, collecttimeout, dbto);
```

Non-Returning Signal Handlers (cont.)

```
    }
}

static void
collecttimeout(timeout)
    time_t timeout;
{
    int save_errno = errno;
    /*
    ** NOTE: THIS CAN BE CALLED FROM A SIGNAL HANDLER. DO NOT ADD
    ** ANYTHING TO THIS ROUTINE UNLESS YOU KNOW WHAT YOU ARE
    ** DOING.
    */
    if (CollectProgress)
    {
        /* reset the timeout */
        CollectTimeout = sm_sigsafe_setevent(timeout,
        collecttimeout, timeout);
        CollectProgress = false;
    }
    else
    {
        /* event is done */
```

Non-Returning Signal Handlers (cont.)

```
        CollectTimeout = NULL;
    }
    /* if no progress was made or problem resetting event,
    die now */
    if (CollectTimeout == NULL)
    {
        errno = ETIMEDOUT;
        longjmp(CtxCollectTimeout, 1);
    }
    errno = save_errno;
}

sm_syslog(level, id, fmt, va_alist)
    int level;
    const char *id;
    const char *fmt;
    va_dcl
    #endif /* __STDC__ */
{
    static char *buf = NULL;
    static size_t bufsize;
    char *begin, *end;
    int save_errno;
```

Non-Returning Signal Handlers (cont.)

```
        int seq = 1;
        int idlen;
        char buf0[MAXLINE];
        char *newstring;
        extern int SyslogPrefixLen;
        SM_VA_LOCAL_DECL
        ... initialization ...

        if (buf == NULL)
        {
            buf = buf0;
            bufsize = sizeof buf0;
        }

        ... try to fit log message in buf, else reallocate it
        on the heap

        if (buf == buf0)
            buf = NULL;
        errno = save_errno;
    }
```

Non-Returning Signal Handlers (cont.)

- the static *buf* could point to a stack buffer (*buf0*)
- if `sm_syslog` interrupted by the signal,
  - after the signal handler uses `longjmp()`
  - $\Rightarrow$ the *buf* will point to an invalid stack address
- example: state change bug in WU-FTPD v2.4

```
/* SIGPIPE handler */
static void
lostconn(signo)
    int signo;
{
    if (debug)
    syslog(LOG_DEBUG, "lost connection");
    dologout(-1);
}

/*
* Record logout in wtmp file
* and exit with supplied status.
```

Non-Returning Signal Handlers (cont.)

```
*/
void
dologout(status)
    int status;
{
    if (logged_in) {
         (void) seteuid((uid_t)0);
         logwtmp(ttyline, "", "");
#if defined(KERBEROS)
              if (!notickets && krbtkfile_env)
                   unlink(krbtkfile_env);
#endif
    }
    /* beware of flushing buffers after a SIGPIPE */
    _exit(status);
}

/* SIGURG handler */
static void
myoob(signo)
    int signo;
{
    char *cp;
```

Non-Returning Signal Handlers (cont.)

```c
/* only process if transfer occurring */
if (!transflag)
    return;
cp = tmpline;
if (getline(cp, 7, stdin) == NULL) {
    reply(221, "You could at least say goodbye.");
    dologout(0);
}
upper(cp);
if (strcmp(cp, "ABOR\r\n") == 0) {
    tmpline[0] = '\0';
    reply(426, "Transfer aborted. Data connection closed.");
    reply(226, "Abort successful");
    longjmp(urgcatch, 1);
}
if (strcmp(cp, "STAT\r\n") == 0) {
    if (file_size != (off_t) -1)
        reply(213, "Status: %qd of %qd bytes transferred",
            byte_count, file_size);
    else
        reply(213, "Status: %qd bytes transferred",
    byte_count);
}
```

Non-Returning Signal Handlers (cont.)

```
}
...
void
send_file_list(whichf)
    char *whichf;
{
    ...
    if (setjmp(urgcatch)) {
        transflag = 0;
        goto out;
    }
    ...
}
```

- if `lostconn()` / `dologout()` is interrupted by a SIGURG after setting UID to 0
  - ⇒ the execution could jumps (ABOR case) to other place running with UID = 0

Non-Returning Signal Handlers (cont.)

- other problem: the `longjmp()` target could be invalid
  - the function that calls `setjmp()` must still be on the runtime execution stack at any point where longjmp() is called from
- example: the `process_message` could return on error letting the signal handler set for SIGPIPE

Non-Returning Signal Handlers (cont.)

```
jmp_buf env;

void pipe_handler(int signo)
{
    longjmp(env);
}

int process_message(int sock)
{
    struct pkt_header header;
    int err = ERR_NONE;
    if(setjmp(env) != 0)
    {
        log("user disconnected!");
        err = ERR_DISCONNECTED;
        goto cleanup;
    }

    signal(SIGPIPE, pipe_handler);

    for(;;)
    {
        if(read_packet_header(sock, &header)) < 0)
```

Non-Returning Signal Handlers (cont.)

```
        return ERR_BAD_HEADER;

    switch(header.type)
    {
        case USER:
          parse_username_request(sock);
          break;
        case PASS:
          parse_password_request(sock);
          break;
        case OPEN:
          parse_openfile_request(sock);
          break;
        case QUIT
          parse_quit_request(sock);
          goto cleanup;
        default:
            log("invalid message");
            break;
    }
}
cleanup:
    signal(SIGPIPE, SIG_DFL);
```

Non-Returning Signal Handlers (cont.)

```
        return err;
}
```

- the `longjmp()` could jump to an invalid stack frame

Signal Interruption and Repetition

- signal handler can be interrupted or called more than once
- if a signal handler may be invoked more than once due to the delivery of multiple signals
- $\Rightarrow$ the handler may perform an operation multiple times that is really only safe to perform once
- cases
  - more signals being handled by the same handler function
  - same signal repeated
- configure the signal handler for once mode
  - `SA_ONESHOT` and `SA_RESETHAND` flags

Signal Interruption and Repetition (cont.)

- in Linux handler setting automatically reset to default when the corresponding signal is handled once
- signal handlers interrupted by other signals
    - the handled signal is blocked
    - other signals could also be blocked by setting the blocking mask
- use of library functions within a signal handler
    - there could be functions that are not asynchronous-safe
    - list of asynchronous-safe library functions: `http://man7.org/linux/man-pages/man7/signal.7.html`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Signal Scoreboard

- evaluate whether code is asynchronous-safe: account for the entire state of the program
  - global variables
  - static variables
  - privilege levels
  - open and closed file descriptors,
  - the process signal mask
  - local stack variables
- signal handler scoreboards
  - function name
  - location
  - signal

Signal Scoreboard (cont.)

- installed
- removed
- unsafe library function used
- notes

Bibliography

1. "The Art of Software Security Assessments", chapter 13, "Synchronization and State", pp. ... – ...
2. "The 24 Deadly Sins of Software Security", chapter 13, pp. 205 – 215
3. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'), `http://cwe.mitre.org/data/definitions/362.html`
4. CWE-364: Signal Handler Race Condition, `https://cwe.mitre.org/data/definitions/364.html`
5. "Delivering Signals for Fun and Profit", `http://lcamtuf.coredump.cx/signals.txt`

Bibliography (cont.)

6. "Symlinks and Cryogenic Sleep", `http://seclists.org/bugtraq/2000/Jan/16`