Secure Operating System Design and Implementation
System Calls

Jon A. Solworth

Dept. of Computer Science
University of Illinois at Chicago

February 1, 2011

Part I

System call overview

Overview

- An OS's architecture is determined by its system call interface
- A system call causes
  - the OS kernel to be entered
  - turns on the privilege bit and thus
  - enables privileged instructions to be executed
- But this is just the mechanics ...

Abstraction

- The syscalls defines a set of abstractions
- The abstractions that an OS provides are relatively high level
- There is no requirement that they be universal, for example that they can support any network protocol
- And hence the OS has semantics
- And that semantics impacts security
- And programming
- And other properties of the system

Impact

- In Ethos, our focus is on security
- In particular, the security of applications
- Which are not part of the OS
- But which are influenced by the OS
- So it is important to ask:
  - How is security affected by the OS?
  - And what can be done to improve security?

Security

- Highly trusted software has to be carefully designed and analyzed (so that it does the right thing)
- All software needs to be minimally authorized (so that the harm it can do is minimize)
- Authenticated (to know what to trust)
- Isolate by default (authorize that which reduces isolation)
- Prevent security holes

Definition

A security hole is a

1. Bug
2. Which can be triggered by an attacker
3. To violate the security specification of the system

Eliminating security holes

- Every security hole starts as a bug
- Eliminating bugs eliminates security holes
- We can try to find bugs and fix them
- But better if we get rid of whole classes of bugs
- Making it easier to reason about programs
- And ultimately to lower complexity of programming

Definition

A pitfall is the semantics of an interface which can result in a security hole.

How can we systematically eliminate pitfalls?

Programming language pitfalls

Programming languages are a source of pitfalls which lead to security holes. Programming languages should be:

- Type safe: so that types are never violated (e.g., buffer overflow)
- Memory safe: so that the map of variables to memory is consistently maintained.
- Integer overflow safe: so that addition of two positive numbers don't result in a negative number
- No explicit concurrency: threads enable race conditions and many other problems.
- Modular: minimize interaction the programmer needs to consider
- Exceptions: to ensure errors are not overlooked

OS pitfalls

- Race conditions: note that OS is inherently parallel as it deals with the outside (parallel) world.
- Well behaved (self-synchronizing) abstractions are desirable
- Prevent weird interleavings: (e.g., pipe semantics)
- Prevent confusing semantics: (e.g., symbolic links)
- Prevent TOCTTOU (Time-of-check-to-time-of-use) errors (provide atomic operations)
- Monolithic semantics (resulting in over privileging and large attack surface)
- Loose authorization over-privileging processes
- Lack of authentication
- Semantics variants

Clarity

Semantic variants leads to a number of problems:

- Unclear semantics
- To many different mechanisms for the same purpose
- Needlessly complex mechanisms (e.g., the complexity to use cryptography)
- Simple error conditions
- Simplify when error can occur
- Avoid standards ambiguity (e.g., undefined parameter order evaluation)

Part II

Ethos system calls

Ethos built-in security facilities

- Very strong authorization
  Information flow, executable, separation of duty, groups
- Authentication
  Built in mechanisms for network authentication (including digital signature)
- Cryptography
  Implicitly managed (e.g., encrypted file system)
- Service based
  Configuration is service based, enabling system to point to service information

Ethos clarity

- Simplified networking
- Type-safe file system and communication
- Concurrency is external to processes
  no signals, threads, or shared memory
- Simplified failure semantics
  Fewer failures, less failures at inconvenient times
- Transaction
  No explicit locking which has availability and other issues

Per process information

- The user on whose behalf the process executes
- The label of the executable
- The file descriptors
- Process group ID and parent process group ID
- Terminate portal virtual process

Events

Events are handles for asynchronous actions which complete later

- All asynchronous syscalls return an event
- A process can have (issued) multiple asynchronous syscalls
- A process can block on one or more events, waiting for any or all events to complete
- When an event completes and is retired, it returns a status and possibly a value
- Events are identified by and EventId, a 64-bit quantity which is guaranteed to never repeat.

File descriptors

File Descriptors are for the following classes

files	devices	directories	terminate
group sets	IPC	networking	debug

There are 6 default descriptors:

stdin	as in POSIX
stdout	as in POSIX
stderr	as in POSIX
rootDirectory	the root directory and therefore cannot be changed
currentDirectory	the current directory
environmentDirectory	the environment directory

No signals

- Signals are a hodgepodge of different things
- They add in asynchrony (poorly) into a process
- But they also add concurrency into the process
- where it has no business being
- We add in asynchrony via events
- Concurrency happens between processes

Virtual processes

- A virtual process is a process per user, created on demand from a fixed executable
- It is created (if it does not already exist) by sending a file descriptor to it
- It solves the problem of authentication
- Have a network connection to a virtual process
- Have a login connection to a virtual process
- No process ever changes the user with whom it is associated

Portals

- A **portal** is a handle to access some process functionality
- It can be used to debug or to terminate a process
- A portal is a file descriptor (and thus protected by authorization)
- Terminate portal can
  - Check ps-style statistics
  - Kill a process
  - Check whether a process exists
  - Get the process groups associated with a process
- Debug protocol
  - Interfere with processes in controlled ways
  - this is the other essential capability of signals

Process groups

- Process groups are nested
- So that each process group (except the leaf process) is composed of lower level process groups.
- Process groups are useful for sets of processes which are used for a common task
- Process groups are created by `fork`
- Process groups are used by processes which contain terminate portals

Authorization

- executable and user **both** are factors in determining permissions
- information flow to preserve confidentiality and to protect integrity
- group mechanism which
  - ensures relative structure between groups
  - controls how members are added and removed from groups
- separation of duty and chinese wall

Transactions

- Transactions span multiple system calls
- Ensure that actions are atomic
- Simplify recovery
- Simplify failure semantics

Types

- In Ethos, files, networking, and IPC are typed
- Applications never need to deal with raw byte streams
- Problem plagued conversions from raw bytes to type date are done in applications
- IPC/Networking use RPC

Part III

Syscalls by category

Categories

Process	create and manage processes
Events	manage events
Files	read and write files
Directories	the name space in which files exist
IPC/Networking	a IPC and networking are unified
Terminate portal	portal for abnormal termination of processes
Debug portal	portal for debugging operations
Authentication	authentication at the console
Transaction	a syscall transaction mechanism

Notation

Describes what the syscall does

$[r_0, r_1, \ldots, r_n] \quad \leftarrow \quad \mathbf{n}(p_0, p_1, \ldots p_m)$

- **n** is the name of the syscall
- $p_0, p_1, \ldots p_m$ are the parameters
- $r_0, r_1, \ldots r_n$ are the return values
- C-binding

$$r_0 = \mathbf{n}(p_0, p_1, \ldots p_m, *r_1, *r_2, \ldots * r_n)$$

Syscall parameters

Name	Type	Description
status	uint	Result of system call
e	EventId	Event identifier
user	string	user name
fd	Fd	file descriptor
pid	ProcessId	Process ID
pgid	ProcessId	Process Group Id
tag	string	tag used to label objects
retirePair	EventRetire	value returned on evaluating an event
time	Time	time
fromMachine	string	from machine
toMachine	string	to machine
service	string	service
virtualProcess	string	virtualProcess
name	string	name of a file or directory
permission	string	access permissions process wants

Process syscalls

create a process.
Fork returns a file descriptor to debug the child process which allows the parent to
(1) communicate with the process to obtain termination informat
(2) debug the process/process termination.
***level* describes the process group level.**
0 for no change
$i > 0$ **for change level i through MaxProcessGroup of process group to the forked process PID**

$[status, debug] \quad \leftarrow \quad$ **fork**$(level)$

Process Groups

- There are at most MaxProcessGroups associated with each process
- In practice, there can be less because duplicate entries reduce the number of process groups
- Values are replace from level *i* through MaxProcessGroups
- Consider the process groups as a stack

1	Bottom of stack (least recently entered data)
...	
MaxProcessGroups	Top of stack (most recently entered data)

- As such, process groups create a hierarchical structure

Process syscalls (cont'd)

change the executable of a process to that of the *fd*
[*status*] ← **exec**(*fd*)

normal termination
[] ← **exit**()

get process's ID
[*pid*] ← **getPid**()

get process's user
[*user*] ← **getUser**()

set terminate portal virtual process
[*status*] ← **setTerminatePortal**(*label*)

Fork/exec

```
ProcessId parentPid = getPid();
status = fork(0, &debug);
ProcessId pid = getPid();
if (parentPid != pid) { // child
  close(debug);

  exec(fd); // must have previously opened
  ...
  exit();
} else { // parent
  ...
}
```


Events

Blocks until event tree is satisfied (see events document)
[*status*] ← **block**(*EventTreetree*)

Blocks on event and the retire event (see events document)
[*status*, *returnPair*] ← **blockAndRetire**(*eventId*)

beep at *time* from epoch
[*status*, $e_{\langle\rangle}$] ← **beep**(*time*)

e must be a completed event
[*status*, *retirePair*] ← **retire**(*e*)

Events

cancel asynchronous event		
[*status*]	←	**cancel**(*e*)
returns vector of completed EventIds		
[*status*, *eventId*[]]	←	**getCompletedEvents**()
returns vector of uncompleted EventIds		
[*status*, *eventId*[]]	←	**getPendingEvents**()

Event data structures

```
// strings denoted with size/ptr
typedef struct {
    msize_t          size;
    void            *ptr;
} MemStruct;

// events return MemStruct or Fd
typedef struct {
    MemStruct       memStruct;    // string values
    Fd              fd;           // file descriptor
} RetirePair;
```


Timer example

```
Status
tsleep(Time t)
{ // sleep for specified number of nanoseconds
  Time time = getTime();

  time = timeAdd(time, t); // library routine

  status = beep(time, &eventId);

  status  = blockAndRetire(eventId, &retirePair);

  return status;
}
```


Filesystem

read the (entire) contents of the file		
[$status, e_{\langle result \rangle}$]	←	**read**(*fd*)
write string to the file		
[$status, e_{\langle\rangle}$]	←	**write**(*fd*, *string*)
get the fileInformation of the file		
[$status, e_{\langle fileInformation \rangle}$]	←	**fileInformation**(*fd*)
Release the fd for the process		
[$status, e_{\langle\rangle}$]	←	**close**(*fd*)
synchronize written files of a process to disk		
[$e_{\langle\rangle}$]	←	**sync**()

Files

- File contain a single typed value
- But that value is for a high level language
- File operations read the current value or
- write a new value
- No seek
- No file locking
- No streams!

Directories

Create a directory in dirFd with name name and label label

$[status, e_{\langle fd \rangle}] \leftarrow$ **createDirectory**$(dirFd, name, label)$

Create a file in dirFd with name name and label label

$[status, e_{\langle fd \rangle}] \leftarrow$ **createFile**$(dirFd, name, label)$

Open a subdirectory of dirFd with name name and permissions p

$[status, e_{\langle fd \rangle}] \leftarrow$ **openDirectory**$(dirFd, name, permissions)$

Open a file of dirFd with name name and permissions permission

$[status, e_{\langle fd \rangle}] \leftarrow$ **openFile**$(dirFd, name, permission)$

Directories (cont'd)

Get the next name greater than name in directory dirFd

$[status, e_{\langle name, type \rangle}] \leftarrow$ **getNextName**$(dirFd, name)$

Remove a directory in dirFd with name name

$[status, e_{\langle \rangle}] \leftarrow$ **removeDirectory**$(dirFd, name)$

Remove a file in dirFd with name name

$[status, e_{\langle \rangle}] \leftarrow$ **removeFile**$(dirFd, name)$

Get file information for dirFd

$[status, e_{\langle fileInformation \rangle}] \leftarrow$ **fileInformation**$(dirFd)$

Directory

- Directory operations don't work on paths, just individual name components
- Directories provide a name space
- Files are just variables
- Directories are streaming
- Can write to a directory
  - Each write creates a separate file indexed by time
  - Ethos has a nanosecond timer (in which successive accesses give monotonically increasing time)
  - Works with concurrent processes
- Hence, directories, IPC, networking are all streaming

Directory/File

```
Status
readVar(Fd dirFd, const char *name,
        MemStruct *memStruct)
{
  Status status;
  status = openFile(dirFd, name, "r", &eventId);
  status = blockAndRetire(eventId, &retirePair);
  Fd fd = retirePair.fd;

  status = read(fd, &eventId);
  status = blockAndRetire(eventId, &retirePair);
  *memStruct = retirePair.memStruct;

  close(fd);
  return status;
}
```


IPC/Networking

equivalent of socket connect

$[status, e_{\langle fd \rangle}] \leftarrow$ **ipc**(*rpc*, *fromMachine*, *toMachine*, *service*)

equivalent of socket bind

$[status, e_{\langle fd \rangle}] \leftarrow$ **advertise**(*rpc*, *toMachine*, *service*)

equivalent of accept

$[status, e_{\langle fd \rangle}] \leftarrow$ **import**(*fd*)

equivalent of accept only from user which owns the process

$[status, e_{\langle fd \rangle}] \leftarrow$ **importUser**(*fd*)

IPC/Networking

is there a new user waiting on the listening socket without a corresponding virtual process

$[status, e_{\langle user \rangle}] \leftarrow$ **newUserWaiting**(*fd*, *virtualProcess*)

Create if necessary virtual process owned by user. Send it the fd.

$[status] \leftarrow$ **fdSend**(*fd*, *user*, *virtualProcess*)

receive fd owned by user which owns process

$[status, e_{\langle newfd \rangle}] \leftarrow$ **fdReceive**()

IPC/Networking

- To unify IPC with networking, several things are needed:
  - Authenticate network connections (cryptographically)
  - Authorize network connections
  - Authentication of IPC is much cheaper
  - IPC authentication by process credential

IPC/Network usage (server)

- Bind equivalent

```
[status, e] = advertise(rpc, toMachine, service)
[status, listenFd] = blockAndRetire(e);
```

- Traditional accept

```
[status, e]     = import(listenFd);
[status, fd]    = blockAndRetire(e);
```

- Per user accept (uses a virtual process)

```
[status, e]       = newUserWaiting(listenFd)
[status, user]    = blockAndRetire(e);
fdSend(listenFd, user, perUserProcess);
```


IPC/Network usage (client)

- Client code

```
[status, e] = ipc(rpc,
                  fromMachine, toMachine,
                  service);
[status, fd] = blockAndRetire(e);
```


Terminate Portal

get the process groups.		
[*status*, *ProcessId*[]]	$\leftarrow$	**getProcessGroups**(*fd*)
get process status.		
[*status*, *ProcessStats*]	$\leftarrow$	**getProcessStatus**(*fd*)
kill process associated with portal.		
[*status*]	$\leftarrow$	**kill**(*fd*)
does the portal's process still exist?		
[*status*]	$\leftarrow$	**isAlive**(*fd*)

Terminate portal

- Ethos does not have a globally visible process table
- Instead, process state is authorized on a per process basis
- Conceptually, each user has its own **process monitor**
- Which can monitor CPU usage and can kill only the user's processes
- We can also build **application monitors** which can monitor multiple process applications and restart applications (by killing all its processes) and then restarting the process.
- These monitors are solely responsible for implementing process group semantics

Terminate portal use

```
setTerminatePortal(``processMonitor'')
// create a process, which will send a terminat
[status, debug] = fork(2);
```


Process monitor code (highly simplified)

```
while (1) {
   [status, event] = fdReceive();
   status = blockAndRetire(event, &retirePair);
   fd = retirePair.fd;
    if (type(fd)==TerminatePortal)
        {
          getProcessGroups(fd);
          // setup tables
        }
    else if (type(fd)==IPC)
        {
          // listen to ipc for user requests
        }
}
```


Debug portal

- Debug portal allows one process to debug another process
- It can stop, single step, and continue processes
- It can read and modify variables
- It can determine the execution path

Authenticate syscalls

user terminal authentication (blocking)

[*status*] ← **authenticate**()

- Authentication occurs in the Ethos kernel
- It could be password based or cryptographic (e.g., smart card)
- It simply returns success or failure
- Because of virtual processes, no need to ever change user of the process

Transaction

start a new transaction. Transactions are not nested.		
[*status*]	←	**beginTransaction**()
complete a transaction, returns true iff successful. (blocking)		
[*status*]	←	**endTransaction**()
abandon a transaction, undoing the operations		
[*status*]	←	**abortTransaction**()

Transaction example

```
beginTransaction();
   // other system calls

   // transaction conditions don't hold, abort
   if (accountStatus == Closed)
       abortTransaction();

   // other system calls

endTransaction();
```


Part IV

Conclusions

Conclusions

Ethos system calls are

- Low level, in that they are asynchronous
- High level, in that they support:
  - types
  - network authentication and encryption transparently provided
  - strong authorization
  - transactions
- intended to work with a high level programming language (which provides types, exceptions, memory management)
- intended to be used with (different) libraries