Secure Operating System Design and Implementation
Coding

Jon A. Solworth

Dept. of Computer Science
University of Illinois at Chicago

February 1, 2011

Part I

Coding Overview

Overview

- Kernel programming environment is more primitive than userspace
- Standard C library not available in user space
- Different interfaces for memory allocation and I/O, for example
- Very primitive debugging environment (register dump w/ procedure name)
- Run-time errors freezes or crashes the kernel
- User space is not trusted, must carefully check anything from user space
- Need to be very careful!

Caution

- You really need to think about things before you put them in a kernel.
- Of course, when you are developing ideas, you can do trial implementations but that is not on the main copy of the code base
- For a new design, you should write (and keep up to date) a new design document
- All the parts of the OS interact, and it is necessary to think carefully about this interaction

This is the first OS we ever built

- We're very conservative, using the most robust construction techniques we know
- We're coding is C, because that is a well trod path—we'll eventually switch to a real programming language.
- Single processor design (low concurrency)
- Simple OS
- Performance is secondary

Part II

The rules

The rules

- There are a large number of rules when building an OS
- These rules are above the programming language
- They include issues such as avoiding security holes, locking, starvation, deadlock, storage allocation, and data structure
- These rules are checked by people

Checking information from user space

- Anything from user space should be treated with suspicion
- All syscall parameters need to be check to ensure they are well formed
- Also need to check they have suitable permissions
- The same goes for network traffic and for the file system.

User space copies

All pointers which are used to copy data to or from userspace must be checked. Ensures memory is in user space and is allocated.

- Copy to kernel from userspace

```
userspace_memcpy_from(uptr, kptr, s)
```

- Copy from kernel to userspace

```
userspace_memcpy_to(uptr, kptr, s)
```

where

`uptr` is the user space pointer,

`kptr` is the kernel pointer, and

`s` is the size in bytes

Integer overflow/underflow

Definition

Integer overflow occurs when addition of two integers is less than either one.

Definition

Integer underflow occurs when addition of two integers is greater than their sum.

The problem is that integer addition is really integer addition modulo 2^s where s is the size of the variable.

Range testing

- Consider the issue of testing whether the numbers $n \in b \ldots b + l$ each satisfy the condition $y \leq n \leq z$.

Range testing

- Consider the issue of testing whether the numbers $n \in b \ldots b + l$ each satisfy the condition $y \leq n \leq z$.
- does this work?

```
if ((b >= y) && ((b+l) <= z))
    OK;
```

Range testing

- Consider the issue of testing whether the numbers $n \in b \ldots b + l$ each satisfy the condition $y \leq n \leq z$.
- does this work?

```
if ((b >= y) && ((b+l) <= z))
    OK;
```

no! $b + l$ can overflow

Range testing

- Consider the issue of testing whether the numbers $n \in b \ldots b + l$ each satisfy the condition $y \leq n \leq z$.
- does this work?

```
if ((b >= y) && ((b+l) <= z))
    OK;
```

no! $b + l$ can overflow

- How about this?

```
if ((b >= y) && (l <= (z-b))
    OK;
```


Range testing

- Consider the issue of testing whether the numbers $n \in b \ldots b + l$ each satisfy the condition $y \leq n \leq z$.
- does this work?

```
if ((b >= y) && ((b+l) <= z))
    OK;
```

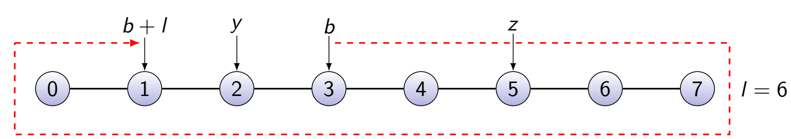
no! $b + l$ can overflow

- How about this?

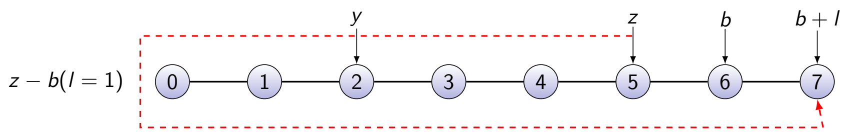
```
if ((b >= y) && (l <= (z-b))
    OK;
```

Sounds like a homework problem

Overflow



Underflow



Bounded buffer example

- Xen's bounded buffer uses free-running indices in which pointers (unsigned integers) into the buffer are always incremented and use there full word size range.
- to insert an element into the buffer it computes $last++$ and then access the buffer using last modulo the buffer size
- to remove an element form the buffer, it access the buffer using first modulo the buffer size and then computes $first++$
- to determine the number of elements in the buffer it computer $last - first$.
- is this correct?

Buffer overflow

- Languages such as C/C++ do not do bounds checking, hence

```
char name[100];
for (i=0; i<n; i++)
    name[i] = '0';
```

- has a buffer overflow if $n > 100$
- there are many variants of this, such as format strings, etc. which one needs to be careful about
- also need to be aware of negative offsets (perhaps from buffer overflow, etc.)
- pointer arithmetic also has this problem

Null pointers

- De-referencing a NULL pointer causes a crash
- Every procedure should check its parameters, e.g.,

```
ASSERT(ptr);
```

- Should check other parameter conditions which must hold
- Should check returns of functions called
- Most procedures coded to return errors (Status), see status.h
- Should check any other relationships that should hold

The purpose of checks

- The purpose of these checks is to determine when assumptions are being violated
- This either indicates a flaw in programming (the easier case) or
- A flaw in the set of assumptions being used
- In any event, the high level structure of the program is broken
- And it is important to know about this as soon as possible

Error, Fault, Failure

Error the problem which results in a failure (bug)

Fault the place where the subsystem behavior deviates from specification

Failure the place where the system deviates from specifications

- Goal in debugging is to find the error
- But the process begins with a failure
- And then trace back to the source (typically through a binary search)
- By placing checks earlier in the code, failures occur faster, and trace back is easier
- (It is also possible in well designed and tested systems to do fault tolerance, detecting faults early and then rolling back state and retrying)

Part III

Kernel Specific Issues

Stacks

- Each process has a kernel stack which is used when the process is executing in kernel mode
- Kernel stacks are small (2 pages), and hence must be careful of stack overflow
- Don't allocate large arrays on the stack (i.e. don't declare large arrays in procedures)
- But we can't allocate them statically either, because we need re-entrancy
- Hence, they should be allocated per call using slob allocator

Floating point

- Ethos does not save floating point registers on entering the kernel
- Therefore, the kernel shouldn't use floating point operations
- The only place where this might be used in cryptographic software

Memcpy/Strcpy

- `strcpy` assume a NULL terminated string
- `memcpy` takes the exact size to copy
- `memcpy` is far more heavily used in the kernel, because the kernel needs to copy the raw bits of data created/used by applications.

Static variables

- Consider:

```
int p(void)
{
  static char *name;
  // ... change name ...
  q(name);
}
```

- This is a problem with re-entrancy, when a process waits on an event
- Consider when `q` can wait
- Then a another process calls `p` and updates `name`
- Now the first processes name is updated too
- Its better to just not use static local variables

Part IV

Test suites

Test suites

- It is important to build automated test suites
- because they are easy to run, you can run them often
- because they describe the errors, it is easy to tell if bugs were introduced
- if you changed N lines of code, and you get an error, most likely the error was introduced in those N lines. Better if $N = 3$ then $N = 1200$.

Easy test suites

- The easiest was to build a test suite is to use two phases
- The first phase runs each test storing the results to output files
- The second phase compares each test run against a known good output
- A very brief output if test is OK, more verbose otherwise
- Very fast visual scan suffices
- Avoid over automation, lets see it work

what should be tested

- Make lots of simple tests
- Build up and make more complicated tests
- Stress testing, throwing random stuff at the OS and see if problems develop
  (stress testing makes it more difficult to determine what is correct output)

Part V

Source control

Source control

- We use subversion to manage source repositories
- Subversion is designed for concurrent developers
- Subversion provides the following benefits:
  - Central place to keep latest good copy
  - Conflict detection and resolution
  - History of changes
  - Backup
  - Integration with tools
- We use it for papers, proposals, etc.—not just for code
- Checking into subversion directory only code that passes test suite

Subversion commands

- Checking out or getting a copy of a repository:
  `svn co svn+ssh://rites.uic.edu/home/svn/projects/ethos/ethos`
- Then edit your local copy
- Add a new file `x`
  `svn add x`
- Update local copy with changes made to repository by others
  `svn update`
- Put changes back to repository (and provide a comment as to change)
  `svn commit`

Layout of source files

trunk Main copy of the source base

branch A temporary copy for long term development separate from the trunk

texnotes documents associated with the project

Source code layout of Ethos kernel

Some directories

xen Xen interfaces

arch/x86 Architecture specific directory (there is some 64-bit code, but that is from MiniOs)

rpc Remote procedure calls, used both in kernel and Dom0 shadow daemon

userspace process level stuff, further divided into `dom0` and *ethos*

include include files which are

- **userspace** both in the kernel and in userspace
- **ethos** only used in the ethos kernel
- **xen** include files from Xen

Part VI

Comments

Comments

- Comments play an important part in understanding the code
- Its primary purpose is to explain higher level structure
- (Documents describe the highest level, this is the next level down)
- Comments should help the reader understand overall structure
- Things to comment: procedures, parameters, files

Commenting tricky code

- Before commenting tricky code, the question should be asked:
- Can this code be simplified?
- Simpler code easier to understand, test, and integrate; less likely to have errors
- Simplification—removal of unnecessary complexity—is the most valuable of programming tasks

Part VII

Coding Rules

Coding rules

A software project should look as if it was coded by a single person

- A consistent style should be used throughout
- Style meant to enhance reading and comprehension of code, eliminate mistakes
- Don't need to comment what is clear from the code, conventions
- Maximize the work that the programming language is doing to clarify, isolate, and describe
- Naming, typing, partitioning into files, etc. all contribute to this

Types

- An OS will be targeted to different architectures
- Some of these will have different memory architectures, including different sized address spaces
- For example, x86 supports both 32 and 64 bit address space
- Types can help bridge the gap between these systems
- Enabling the OS to be split into architectural dependent code and
- architectural independent mode which uses only types to distinguish between actions.

Prominent "primitive" types

int	32-bit on either 32-bit or 64-bit architectures
long	the word size of the architecture (32 or 64 bit)
uint32	unsigned integer of 32 bits
uint64	unsigned integer of 64 bits
int32	signed integer of 32 bits
int64	signed integer of 64 bits
vaddr_t	virtual address (as an unsigned integer)
msize_t	an unsigned integer large enough to index memory
paddr_t	physical address

Naming

- Names of procedures should have the file name as a prefix
- Use camel case names, e.g., `aDogBitMe`
- Procedures and variables start with lower case
- Types and Constants start with upper case

Naming

- Types, even if they are the same as a primitive type, should bit `typedefed` so that the type says what they are used for. E.g., `addr` is an `unsigned long` but is used when value is an address.
- avoid unnamed constants (magic constants) in code, better to name them and then use them
- procedure names with `alloc` in them allocate storage, with `create` allocates and initializes storage.
- Enums are used in preference to const. Enums have the advantage over const that duplicates are OK. They have the advantage of `#define` that they are not macros
- There should be a very good reason to code any macro in Ethos

Include files

- Include files should center around one thing
- To ensure files only loaded once per .c file, use the following form for file `xY.h`

```
#ifndef __X_Y__
#define __X_Y__
  // contents here

#endif
```


Indentation

```
Status
p(char *name, ///<  name of variable to be output
  int n       ///<  number of bytes used in name
  )
{
    int i;
    for (i=0; i<n; n++)
        {
            output(name[i]);
        }
    return StatusOk;
}
```