Hydra: Non-hierarchical Protection Course Wrapup

Arvind Krishnamurthy
Spring 2004

Hydra

- Motivation was C.mmp multiprocessor project:
  - First multiprocessor with 16 PDP-11 machines
  - Hydra is the OS designed for this system
  - Didn't know what kind of functionality for parallel machines
    - Experiment with new kinds of "subsystems"
- Micro-Kernel system: minimal OS
  - Have a small piece of code to coordinate subsystems
  - Most of the functionality moved to subsystems
- Lots of subsystems that run at user-level
  - Debug them like normal user programs
  - Have several alternative for a subsystem running simultaneously

Overall Design of Hydra

- Subsystems: code and data (protected)
  - Only the code in the subsystems can access the data
- Non-hierarchical protection
- What is required to support non-hierarchical protection?
  - Seal objects: subsystems seal objects before passing it to other subsystems
  - To manipulate, return to subsystem, unseal

User Extensible Types & Capabilities

- Objects comprise of:
  - Name: unique number
  - Type: the id of the type from which object was instantiated
  - Representation comprising of data and capabilities
- Capabilities:
  - Object name
  - Permissions: what operations can be performed

Built-in Types

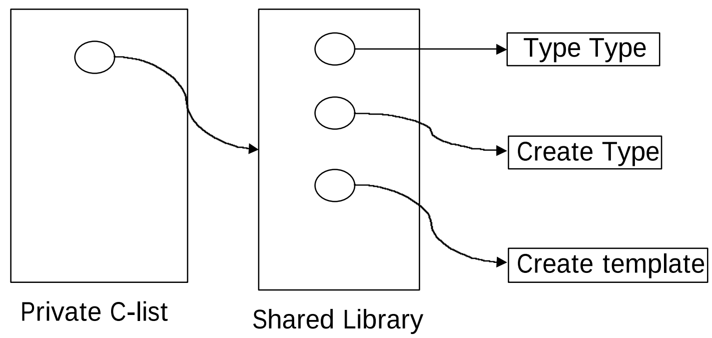
1) Page: map into address space
2) Procedure: static information
   Code, static data
3) Local name space (LNS)
   Activation record
   Contains a pointer to procedure object, arguments, and working space
4) Process: stack of LNSs
5) Template object (discuss next)

Sealing/Unsealing

- Template contains:
  - Old privileges, new privileges
  - Kernel checks whether argument matches template
  - Procedure gets new capability with new privileges
  - Think of it as rights amplification
  - Template created from type object

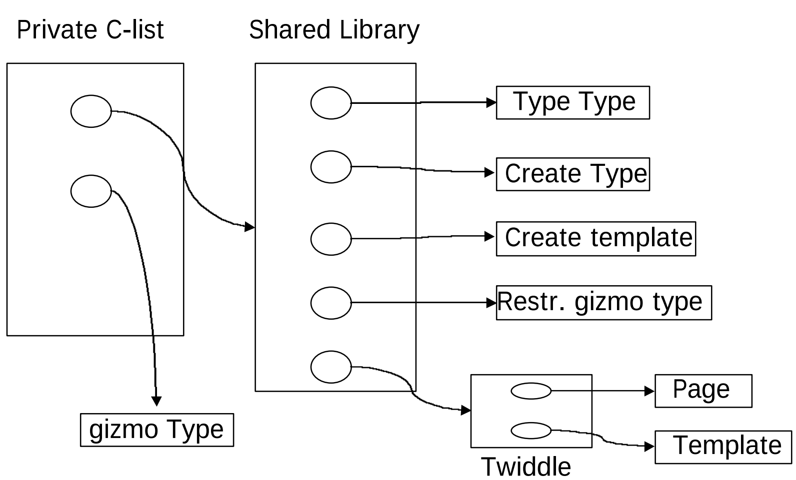
Sample Interaction

- When a person logs in, he/she gets a capability list of useful capabilities



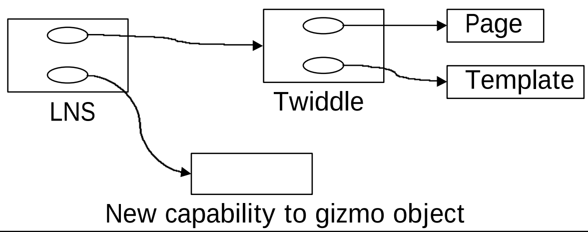
Create New Type

- Creating type (called gizmo) and new procedures (twiddle) that can operate on gizmo



Making Inter-subsystem Call

- Pass an instance of gizmo object to twiddle procedure
- Kernel does the following using the template object:
  - Checks whether object is indeed of gizmo type
  - Checks whether object has permissions to have twiddle be invoked on it
  - Amplifies rights for the object
  - Creates an LNS for the procedure call
  - Attaches LNS to the process's LNS stack



Hydra Analysis

- Four issues:
  - Cost of subsystem calls
  - Persistent object store:
    - Disk I/O costs
    - Garbage collection costs
  - Kernel size
  - Applications: needed to demonstrate the need for different subsystems

Course Wrapup

- Covered the key components in an OS and in building systems:
  - Synchronization, virtual memory, file systems, networking, security, and distributed systems
  - Studied some great successes and some glorious failures

- Recommended future reading:
  - Mythical Man Month by Brooks
  - Hints for System Design by Lampson
  - Hints for Language Design by Hoare

- Wrapup: some material Lampson's paper, Ken Thompson's turing award lecture and Hoare's turing award lecture

Classes of security problems

- **Abuse of privilege** --- if the super user is evil, we are all in trouble; there is nothing you can do about this.

- **Impostor** --- break into system by pretending to be someone else. (e.g., rhost, insecure X sessions)

- **Trojan horse** --- appears helpful but really harmful

- **Eavesdropping** --- tap onto Ethernet and see everything typed in

- **Salami attack** --- steal and corrupt something a little bit at a time
  (partial pennies from bank interest – plot of Superman 3 – happened in real life)

Concrete Examples: Tenex

- Most popular system at universities before Unix (early 70's)
- Thought to be very secure. To demonstrate it, created a team to find loopholes. Gave them all the source code/doc
- In 48 hours, had every password in the system!

- Code for password check:

```
PasswordCheck(char *userPasswd)
    for (i=0; i<8; i++)
        if (userPasswd[i] != realPasswd[i])
            Goto error;
```

- Looks innocuous, like you'd have to try all combinations

Tenex (contd.)

- Problems arise because of the combination of the following three design choices:
  - Tenex used virtual memory
  - It had a system call for checking passwords
  - In addition, it didn't make a copy of the user arguments on system calls

- How do you break into such a system?

Internet Worm

- 1988: a worm broke into thousands of computers over the internet
  - Apparently initiated by Robert Morris Jr.
- Three attacks:
  - Dictionary lookup
  - Sendmail: debug mode, if configured wrong, can let anybody log in
  - fingerd: “finger arvind@lambda.cs.yale.edu”
    - fingerd didn’t check for length of string
    - Allocated a fixed size array for it on the stack

```
foo(char *s) {
    char buffer[200];
    ...
    strcpy(s, buffer);
}
```

Self-replicating program

- Proposed by Ken Thompson in his Turing award lecture
- Bury trojan horse in binaries, so no evidence in the source
- Replicates itself to every Unix system in the world, and even to new Unixes on new platforms
- Two steps:
  - Make it possible (easy)
  - Hide it (tricky)
- Step 1: Modify login.c (code snippet A)

```
If (name == “ken”)
    Don’t check password
    Log in as root
```

- Next step: hide change, so no one can see it

Modify the C Compiler

- Step 2: Instead of having the code in login.c, put it in the compiler (code snippet B):

  ```
  If see trigger
      Insert A into input stream
  ```

- Whenever the compiler sees a trigger (/* gobbledygook */), puts A into input stream of the compiler
  - Now don't need A in login.c, just need the trigger
  - Need to get rid of the problem in the compiler

```
login.c:

  /* gobbledygook */
```

→

```
Compiler:
  if (str == "gobbledygook")
    emit code for
    trojan horse
```

Modify the C compiler

- Step 3: modify compiler to have (code snippet C)

  ```
  If see trigger2
      Insert B + C into input stream
  ```

  - This is where the self-replicating code comes in!
  - Question: can you write a program that has no inputs and outputs itself (or a superset of itself)?

- Step 4: Compile the compiler with snippet C present
  - Now the intelligence is in the binary

```
Compiler code:

 /* gobbledygook2 */
```

→

```
Compiler binary:
  if (str == "gobbledygook2")
    emit code for trojan check
    and replicate this check
```

Self-replicating program (contd.)

- Step 5: replace snippet C with trigger2
  - Result: all of the intelligence is only in the binary and not in the source code!
- If you use binary to compile “login.c”, it will recognize trigger to emit backdoor
- If you use binary to compile the compiler, it will recognize trigger2
  - It will emit code in the generated binary to watch out for invocations when you are compiling “login.c” or the compiler itself
- Summary: can’t stop loopholes, can’t tell if it’s happened, can’t get rid of it!

Tony Hoare’s Turing Lecture

- Tony Hoare’s accomplishments:
  - Quicksort!
  - Study of monitors
  - CSP language (communicating sequential processes)
  - Axiomatic semantics of programming languages
- Turing Award Lecture:
  - Anecdotes on simplicity and system design
  - Successes: compiler for Algol on Elliot 503
  - Failures: operating system (Mark II for Elliot 503)
  - Experiences from language design committee meetings

The Algol60 Story

- Hoare started as a programmer for Elliot Brothers
  - Implemented Shell sort and other fast routines
  - Designed a variant of Shell sort (which became quick sort)
  - Given the task of designing a new high level language
- In 1961, attended a course on Algol60 by Naur, Djikstra, Landin
  - Was able to implement his sorting variant using recursion
  - Decided to use Algol60 as the high level language for their machines
- Designed and documented in Algol60
  - Then coded in machine language by hand!
  - Using explicit stack for recursion
  - Started with a small subset, was able to add more features later

Language Implementation

- Some great ideas:
  - Security: bounds checking!
  - Brevity of object code
  - Fast language features (such as procedure calls)
  - Single pass compiler (not so important in the current day)

Language Design of Algol 60

- "Awarded" with membership in the Algol language working committee
- Lessons from the design meetings:
  - Avoid complex language features such as overloading of operators, default type conversions, etc.
    - Can either design a system that has "obviously no deficiencies or has no obvious deficiencies … the latter is easier"
  - Avoid generalized "goto" features and other irregular control structures
  - relax compulsory declarations (supposed bug resulting in Mariner rocket crash – which since then is considered a hoax)

```
      DO 10 I = 1.10
       ...
10    CONTINUE
```

Significant failure: OS for Mark II

- Hoare got promoted, and Elliot started building Mark II
  - Assembler
  - Automatic code and data overlays from backing core, tape
  - Automatic input/output buffering
  - Filing system on tape
  - New implementation of Algol60, Fortran compiler
- Designed system, set deadline in 18 months (march 1965)
  - Revised to June, later revised by another 3 months, …
  - Started digging into the details:
    - Limited resource of memory wasn't handled either by the assembler or the automatic overlay scheme
    - System was occupying most of the memory resource!
    - Hardware address limits prevented adding more memory

Failure (contd.)

- Decided to focus on just the Algol60 compiler
  - Revised delivery in 4 months
  - Delivered, but
  - Original compiler: compiled 1000 chars/second, new compiler: 2 chars/second!
  - Problem: thrashing
  - Within a week doubled it, and doubled it in another two weeks
  - Not improving fast enough, project had to be abandoned
- Recovery:
  - Called a meeting to discuss the reasons:
    - Lack of machine time, unpredictable hardware etc., technical writing for documentation
    - Over-ambition – the second system effect?
  - Hoare realized that he shouldn't let others do what he didn't understand himself!

Final comments

- Simplicity: Simplicity is an absolute good, not a tradeoff!
  - Easier to build/maintain, run faster, is cheaper
  - Forces against it:
    - "Complexity = Intelligence", "more features (complexity) is good"
  - How to make things simpler? Creativity and design before coding
- Performance: make every line of code as fast as possible vs. selective tuning (much better) – only a few places where performance matters
  - Biggest gain is going from non-functional to functional! Then add new data structures/algorithms to make the system go faster
- Life as a CS person:
  - Computers are tools: need to understand applications
  - If demonstrating an idea/doing research: Go to the extreme!
  - Stay broad: Breadth helps depth
  - Technology is changing fast; exploit new changes/shifts
  - Breakthroughs occur when people cross traditional boundaries: compilers and architecture, graphics and VLSI, etc.; steal from other fields!