Overview of Language-Based Security

Dan Grossman
590NL
13 October 2004

Note: There is an accompanying bibliography for this presentation.

Why PL?

To the extent *security is a software problem*, PL has a powerful tool set (among several)

- Much progress in last few years
- Practical applications
- Plenty of open and fun core-CS research

(But security is not just a software problem:
encryption, tamper-proof hardware, policy design, social factors, …)

The plan

- 1st things 1st : the case for strong languages
- Overview of the language-based approach to *security* (as opposed to software quality)
- Many examples and pointers

Next week:
UW projects and their connection to security

Just imagine…

- Tossing together 20000000 lines of code
- From 1000s of people at 100s of places
- And running 10000000s of computers holding data of value to someone
- And any 1 line could have arbitrary effect

All while espousing the principle of least privilege?!

Least Privilege

"Give each entity the least authority necessary to accomplish each task"

versus

- Buffer overruns (read/write *any* memory)
- Code injection (execute *any* memory)
- Coarse library access (`system` *available by default*)

Secure software in unsafe languages may be possible, but it ain't because of least privilege

The old argument

- Better languages à better programs à better security
  - Technically: strong abstractions isolate errors (next slide)
- "But safe languages are slow, impractical, imbractical"
  - So work optimized safe, high-level languages
  - Other work built safe C-like languages (me,…)
  - Other work built safe systems (SPIN,…)
  - (and Java started bractical and slow)
- Meanwhile, seminar on TC, not performance

Abstraction for Security

Memory safety isolates modules, making strong interfaces (restricted clients) enough:
Example: Safer C-style file I/O (simplified)

```
struct FILE;
FILE* fopen(const char*, const char*);
int   fgetc(FILE*);
int   fputc(int, FILE*)
int   fclose(FILE*);
```

No NULL, no bad modes, no r/w on w/r, no use-after-close, else "anything might happen"

File Example

Non-NULL (Cyclone)

```
struct FILE;
FILE* fopen(const char@, const char@);
int   fgetc(FILE@);
int   fputc(int, FILE@)
int   fclose(FILE@);
```

Client must check `fopen` result before use

More efficient than library-side checking

File Example

No bad modes (library design)

```
struct FILE;
FILE* fopen_r(const char@);
FILE* fopen_w(const char@);
int   fgetc(FILE@);
int   fputc(int, FILE@)
int   fclose(FILE@);
```


File Example

No reading files opened for writing and vice-versa

Repetitive version:

```
struct FILE_R;
struct FILE_W;
FILE_R* fopen_r(const char@);
FILE_W* fopen_w(const char@);
int     fgetc(FILE_R@);
int     fputc(int, FILE_W@)
int     fclose_r(FILE_R@);
int     fclose_w(FILE_W@);
```


File Example

No reading files opened for writing and vice-versa

Phantom-type version (PL folklore):

```
struct FILE<'T>;
struct R; struct W;
FILE<R>* fopen_r(const char@);
FILE<W>* fopen_w(const char@);
int      fgetc(FILE<R>@);
int      fputc(int, FILE<W>@)
int      fclose(FILE<'T>@);
```


File Example

No using files after they're closed

Unique pointers (Vault, ArchJava, Cyclone 0.8)

```
struct FILE<'T>;
struct R; struct W;
unique FILE<R>* fopen_r(const char@);
unique FILE<W>* fopen_w(const char@);
int  fgetc(FILE<R>@);
int  fputc(int, FILE<W>@)
int  fclose(unique use FILE<'T>@);
```

Moral

- Language for stricter interfaces:
  - Pushes checks to compile-time / clients (faster and earlier defect detection)
  - Can specify sophisticated idioms
  - Incremental adoption possible
- Memory safety a precondition to any guarantee
- But getting security right this way is still hard, and hard to verify
  - "Does this huge app send data from home directories over the network?"

The plan

- 1st things 1st : the case for good languages
  - Some more on Cyclone next week (array lengths)
- Overview of the language-based approach to *security* (as opposed to software quality)
- Many examples and pointers

Language-based *security* is more than good code:

Using PL techniques to enforce a security policy

Summary of dimensions

1. How is a policy expressed?
2. What policies are expressible?
3. What is guaranteed?
4. What is trusted (the TCB) ?
5. How is the policy enforced?

Grain of salt: This list is in alpha-test.

But it should help us talk about lots of good work.

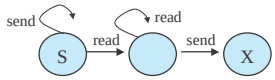
Dimensions of L.B. Security

1. How is a policy expressed?

code:

```
void safesend(){if(disk_read) die();…}
```

automata:



logic:

∀ *states s. read(s) à (forever(not(send)))(s)*

informally: "no send after read"

implicitly: `% CommunicationGuard –f foo.c`

Dimensions of L.B. Security

2. What policies are expressible?

safety properties: "bad thing never happens"
send-after-read, lock reacquire, exceed resource limit

liveness properties: "good thing eventually happens"
lock released, starvation-freedom, termination
- often over-approximated with safety property

information flow
(cf. mandatory/discretionary access control)

confidentiality vs. integrity
(cf. read/write)

Dimensions of L.B. Security

3. What is guaranteed?

Enforcement:
- sound: no policy-violation occurs
- complete: no policy-follower is accused
- both
- neither

Execution:
- meaning preserving: programs unchanged
- "IRM" guarantee: policy-followers unchanged

Dimensions of L.B. Security

4. What is trusted (the TCB) ?

   Hardware, network, operating system, type-checker, code-generator, proof-checker, web browser, …

   programmers, sys admins, end-users, …

   crypto, logic, …

   (less is good)

Dimensions of L.B. Security

5. How is the policy enforced?

static analysis: before program runs
   often more conservative, efficient
   "in theory, more powerful than dynamic analysis"

dynamic/post-mortem analysis: while/after program runs
   "in theory, more powerful than static analysis"

code generation: how code is compiled

environment: libraries, hardware

Summary of dimensions

1. How is a policy expressed?
2. What policies are expressible?
3. What is guaranteed?
4. What is trusted (the TCB) ?
5. How is the policy enforced?

Grain of salt: This list is in alpha-test.

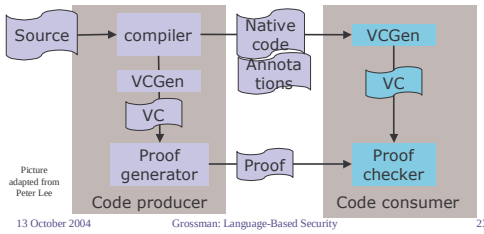
The plan

- 1st things 1st : the case for good languages
- Overview of the language-based approach to *security* (as opposed to software quality)
- Many examples and pointers

  See accompanying bibliography for proper attribution

Proof-Carrying Code

- Motivation: Smaller TCB, especially network
- How expressed: logic
- How enforced: proof-checker



PCC in hindsight (personal view)

- "if you can prove it, you can do it"
  - dodges undecidability
- key contributions:
  - proof-checking easier than proof-finding
  - security without authentication
- works well for many compiler optimizations
- but in practice, policies weak & over-approximated
  - e.g., is it exactly the Java metadata for a class
  - e.g., does it use standard calling convention

Other PCC instances

- Typed Assembly Language (TAL)
  - As in file-example, types let you encode many policies (in theory, any safety property!)
  - Proof-checker now type-checker, vcgen more implicit
  - In practice, more flexible data-rep and calling-convention, worse arithmetic and flow-sensitivity
- Foundational PCC (FPCC)
  - Don't trust vcgen, only semantics of machine and security policy encoded in logic
  - Impressive TCB, > 20 grad-student years

Verified compilers?

- A verified compiler is a decades-old dream
  - I don't think we're close
  - Tony Hoare's recent "grand challenge"
- Why is PCC-style easier?
  - Judges compiler on one program at a time
  - Judges compiler on a security policy, not correctness
- This is little consolation to programmers hitting compiler bugs
- Next week: UW work on optimizations that are provably correct for all source programs

Inline-Reference Monitors

- Rules of the game:
  - Executing P goes through states s1, s2, …
  - A safety policy S is a set of "bad states" (easily summarized with an automata)
  - For all P, the IRM must produce a P' that:
    - obeys S
    - if P obeys S, then P' is equivalent to P
- Amazing: An IRM can be sound and complete for any (non-trivial) safety property S
  - Proof: Before going from s to s', halt iff s' is bad
  - For many S, there are more efficient IRMs

(Revisionist) Example

- In 1993, SFI:
  - Without hardware support, ensure code accesses only addresses in some aligned 2^n range
  - IRM: change every load/store to mask the address

```
sto r1->r2   ⇒   and 0x000FFFFF,r2->r3
                 or  0x2a300000,r3->r3
                 sto r1->r3
```

  - Sound (with reserved registers and special care to restrict jump targets)
  - Complete (if original program obeyed the policy, every mask is a no-op)

Dodging undecidability

How can an IRM enforce safety policies soundly and completely:

- It rewrites P to P' such that:
  - P' obeys S
  - If P obeys S, then P' is equivalent to P
- It *does not decide* if P satisfies the policy

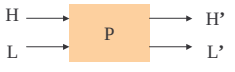
Static analysis for information flow

Information-flow properties include:

- Confidentiality (secrets kept)
- Integrity (data not corrupted)

(too strong but useful) confidentiality: non-interference

- "High-security input never affects low-security output"



∀ H1,H2,L
if P(H1,L) = (H1',L')
then ∃H2'
P(H2,L) = (H2',L')

(Generalizes to arbitrary lattice of security levels)

Non-interference

- Non-interference is about P, *not* an execution of P
  - *not* a safety (liveness) property; can't be monitored
- Robust definition depending on "low outputs"
  - I/O, memory, termination, timing, TLB, …
  - Extends to probabilistic view (cf. DB full disclosure)
- Enforceable via sound (incomplete) dataflow analysis
  - L ≤ H, assign each variable a level `sec(x)`

`e1 + e2`	`x := e`	`if(x) e;`
`max(sec(e1), sec(e2))`	`sec(e) if sec(e)≤sec(x)`	`sec(x) if sec(x)≤sec(e)`

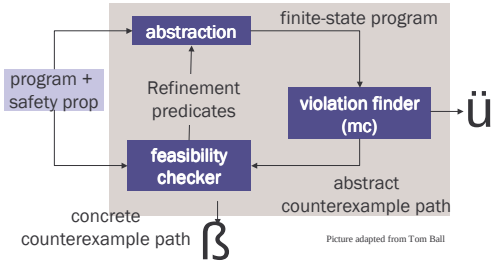
Information-flow continued

`e1 + e2`	`x := e`	`if(x) e;`
`max(sec(e1), sec(e2))`	`sec(e) if sec(e)≤sec(x)`	`sec(x) if sec(x)≤sec(e)`

- *Implicit flow* prevented, e.g., `if(h) l:=3`
- Conservative, e.g.,
  `l:=h*0; if(h) l:=3 else l:=3`
- Integrity: exact same rules, except H ≤ L !
- One way to "relax noninterference with care" (e.g., JIF)
  - `declassify(e)` : L e.g., average
  - `endorse(e)` : H e.g., confirmed by > k sources

Software model checking

Predicate-refinement approach (SLAM, Blast)



Software model checking

- Sound, complete, and static?!
  - (That picture has a loop)
  - In practice, the static pointer analysis gives out first with a "don't know"
  - For model-checking C, typically make weak-but-unsound memory-safety assumptions
- Predicate-refinement just one approach (see bibliography for others)

Metacompilation

Write application-specific checkers in terms of code patterns and automata

+ Define checkers without being a compiler writer
− Unsound (aliasing, memory safety, …)
− Syntactic patterns ensure incompleteness w.r.t. the semantic policy you care about
+ Well-designed to reduce false positives and rank potential violations
+ Bugs get found and fixed (1000s in Linux)

Well-designed extensible bug-finders are great, but they never ensure a policy is obeyed

So far: more general approaches

- PCC
- IRM
- Information Flow
- (Refinement-Based) Model Checking
- Extensible Bug-Finding

Many other tools/techniques have narrower goals and are correspondingly easier to use…

lint-like tools, confinement, stack inspection, type qualifiers, …

Push-button bug-finding

- Prefix, Splint (LCLint), MS Office annotations, etc.
- Does nothing for malice
- Extensibility less important than turning off features, minimizing false positives (ideally complete), and prioritizing results
  - Typically based on “smelly syntactic patterns”
- Dynamic counterparts: Purify, etc.

Confinement

Stronger than private fields, weaker than confidential

- Captures a useful idiom (copy to improve integrity)
- Shows private describes a field, not a value
- Backwards-compatible, easy to use

```
private Identity[]
signers;
…
public Identity[]
getSigners() {
//breach: should copy
  return signers;
}
```

```
confined class
ConfIdent{ … }
ConfIdent[] signers;
…
public Identity[]
getSigners() {
 // must copy
}
```


Java Stack Inspection

- Methods have *principals* (e.g., code source)
- Principals have *privileges* (e.g., file-delete)
- Operations:
  - **`BeginPrivilege();`** // use “my” privileges (set a bit in the call frame)
  - **`CheckPrivilege(P);`** // check for privilege P (start at current stack pointer, find first frame with bit set; check the frame’s method’s principal’s privileges)

\+ principle of least privilege (default is less enabled)

− unclear what the policy is

Stack Inspection Examples

```
delete_file(String s){
  CheckPrivilege(file_delete);
  …
}

util(Object evil) {
   … do not call BeginPrivilege() …
   evil.m(); // safe, unknown callback
}

delete_tmp_a() {
  BeginPrivilege();
  delete_file(“/tmp/a”);
}
```


Type Qualifiers

- Programmer defines qualifiers…
  - e.g., locked/unlocked
- and how “key functions” affect them
  - e.g., **`locked mtx acquire(unlocked mtx);`**
- A scalable flow-sensitive analysis ensures the qualifiers are obeyed
  - e.g., **`acquire`** never passed a **`locked mtx`**
- Precision typically limited by aliasing
- Other published uses:
  - const-inference, tainted-strings, …

Take-Away Messages

- PL has great tools for software security; good languages are *just* a start
- Many approaches to what to enforce and how
- A hot area, with aesthetically pleasing results and practical applications
- As always, learning related work is crucial
- This was 100 hours of material in 45(?!) minutes:
  - see the bibliography and come talk to me