

## Lecture 3: Designing and Writing Secure Application Code

ECE1776  
David Lie

1

## General topics in this Lecture

- Last lecture:
  - Vulnerabilities and how they are exploited
  - How to prevent buffer overflows
- Defensive programming
  - Build *all* software with security in mind
  - Reliability and Security have common goals
- Techniques to create secure code
  - Verification techniques find bugs

} This lecture



2

## Software Engineering 101

- Building secure code is basically building **good** code
  - As we saw in last lecture, all security vulnerabilities are a result of flaws, which are a result of mistakes made by humans
  - The fewer flaws, the more secure the code will be
- Thus to build secure code, we can follow standard software engineering practice:
  - However, since security sensitive vulnerabilities can be more damaging than normal vulnerabilities, we want to pay more attention to practices that will reduce the number of vulnerabilities rather than just flaws overall



3

## Before you start building ...

1. Know what the security requirements are (can be any or all of):
    - Confidentiality (secrets remain secret)
    - Integrity (meaning preserved)
    - Availability
    - Accountability
  2. What threats are possible?
    - Know what the attackers capabilities are
  3. Who do you trust / not trust?
    - Know what inputs/interfaces can be trusted, which ones are suspect
- **Better to be conservative/paranoid on all of the above than be sorry!**



4

## General Rules of Thumb

1. Keep it Simple (Stupid)
  - Probably the most important rule in software engineering
  - The larger the number of interacting factors, the greater chance of a flaw
2. Compartmentalize / Defense in Depth
  - Have many layers of security, remember attacker will exploit the weakest link
  - Keep each layer independent, so that flaws in one are caught by the next
  - If you have to fail, fail in a way that is secure, do not leave the system in an undefined state
3. Principle of Least Privilege / Be Reluctant to Trust
  - Give each actor as much information and ability as they need to do their job



5

## General Rules of Thumb

4. Information Hiding is Hard / Promote Privacy
  - Keep the minimum information around (trash searching)
  - Be ready if information is leaked (think ahead about revocation)
3. Follow standard practice / Use Community Resources
  - Don't try to reinvent the wheel, consult experts and follow standards (especially when it comes to cryptography)




6

### Example: Mail Transport Agents

---

- Sendmail
  - Routing program used to send mail between hosts on the internet
  - Released Circa 1983
  - Uses SMTP (Simple Mail Transfer Protocol) to route mail between agents
- Has poor security properties. The protocol itself does little authentication, but is also complex making it hard to implement
  - SMTP was designed to be human readable, but as a result is very hard to parse
  - Requires root privileges to deliver mail
  - Has resulted in [fourteen security holes](#), discovered in sendmail in 1996 and 1997.



7


### Simplified Mail Transactions

---

```

    graph LR
      subgraph Local
        MUA1[Mail User Agent] --> MTA1[Mail Transport Agent]
        MTA1 --> MDA1[Mail Delivery Agent]
        MDA1 --> mbox1[mbox]
      end
      MTA1 <--> MTA2[Mail Transport Agent]
      subgraph Remote
        MTA2 --> MUA2[Mail User Agent]
        MUA2 --> mbox2[mbox]
        mbox2 --> MDA2[Mail Delivery Agent]
        MDA2 --> MTA2
      end
  
```

- Message composed using an MUA
- MUA gives message to MTA for delivery
  - If local, the MTA gives it to the local MDA
  - If remote, transfer to another MTA




8

### Case Study: qmail

---

- Written by Dan Bernstein
  - Starting in 1995 Bernstein has a reward for \$500 for anyone to find a vulnerability in qmail, no one has collected
- Qmail compartmentalizes tasks for simplicity
  - Nine separate modules
  - Each runs under different non-privileged UID: qmaild, qmailr, qmailq, ... (except one as root)
  - If one module compromised, others not
    - SMTP server qmail-smtpd runs as user qmailr
    - Rest of the system runs as other users




9

### Example: qmail

---

- Least privilege
  - Each module uses least privileges necessary
  - Qmail only has 3 uid's
  - Only one setuid program
    - Purpose is to add mail to the outgoing queue
    - setuid to one of the other qmail user IDs, not root
    - No setuid root binaries
  - Only one program runs as root
    - Spawns the local delivery program under the UID and GID of the user being delivered to
    - No delivery to root
    - Always changes effective uid to recipient before running user-specified program




10

### Keep it simple

---

	Lines	Words	Chars	Files
qmail-1.01	16028	44331	370123	288
sendmail-8.8.8	52830	179608	1218116	53
zmailer-2.2e10	57595	205524	1423624	227
smail-3.2	62331	246140	1701112	151
exim-1.90	67778	272084	2092351	127




11

### Structure of qmail

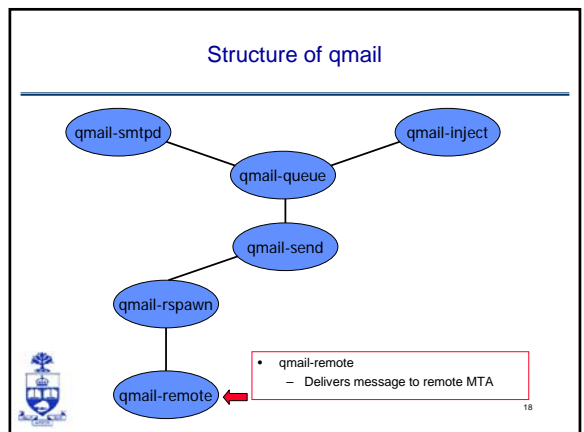
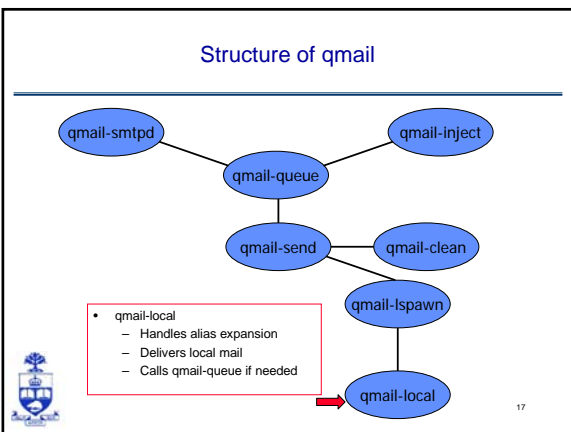
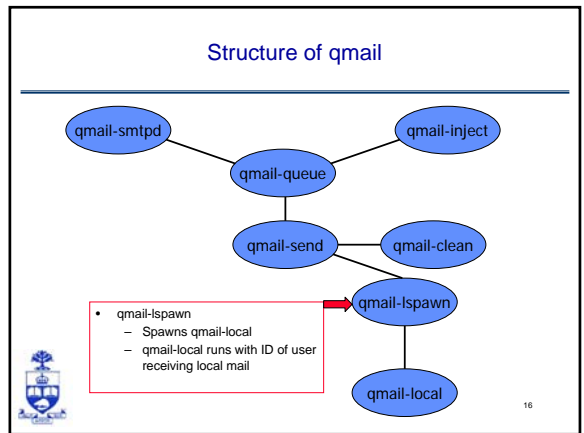
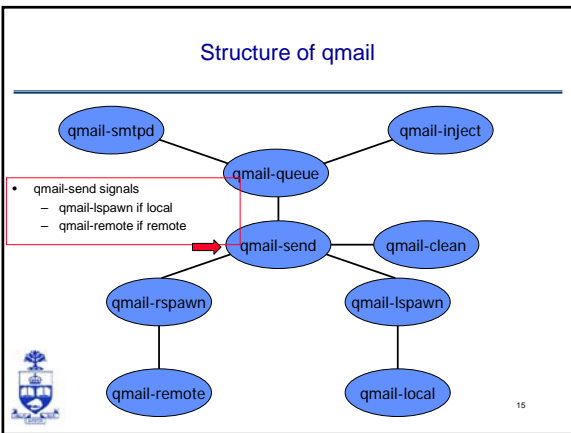
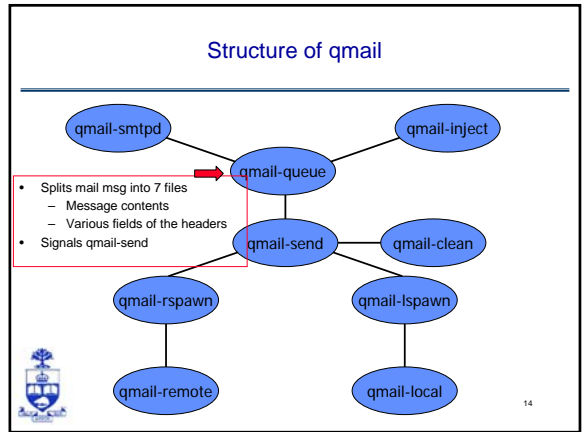
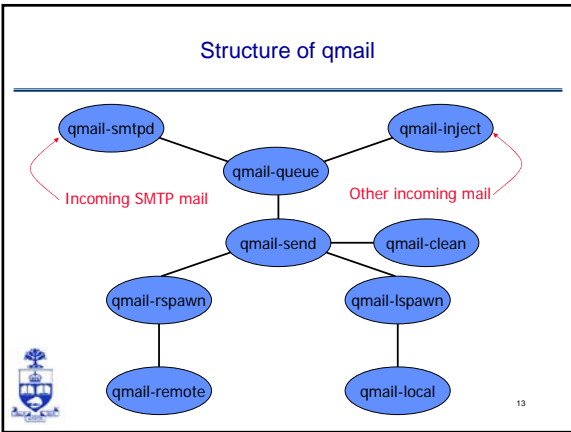
---

```

    graph TD
      qmail-smtpd --> qmail-queue
      qmail-inject --> qmail-queue
      qmail-queue -- setuid --> qmail-send
      qmail-send --> qmail-clean
      qmail-send --> qmail-rspawn
      qmail-send --> qmail-lspawn
      qmail-rspawn --> qmail-remote
      qmail-lspawn -- root --> qmail-local
  
```



12



## Principles: least privilege

- Do as little as possible in setuid programs
  - Of 20 recent sendmail security holes, 11 worked only because the entire sendmail system is setuid
  - Only qmail-queue is setuid
    - Its only function is add a new message to the queue
- Do as little as possible as root
  - The entire sendmail system runs as root
    - operating system protection has no effect
  - Only qmail-start and qmail-lspawn run as root.



19

## Principle: keeping it simple

- Programs and files are not addresses
  - sendmail treats programs and files as addresses
    - "sendmail goes through horrendous contortions trying to keep track of whether a local user was responsible for an address. This has proven to be an unmitigated disaster"  
(DJB)
  - qmail programs and files are not addresses
    - "The local delivery agent, qmail-local, can run programs or write to files as directed by ~user/.qmail, but it's always running as that user. Security impact: .qmail, like .cshrc and .exrc and various other files, means that anyone who can write arbitrary files as a user can execute arbitrary programs as that user. That's it."  
(DJB)



20

## Keep it simple

- Parsing
  - Limited parsing of strings
    - Minimizes risk of security holes from configuration errors
- Libraries
  - Avoid standard std C library, write your own functions
    - "Write bug-free code" (DJB)
- Don't repeat functionality
  - One simple mechanism handles forwarding, aliasing, and mailing lists (instead of 3)
  - Single delivery mode instead of a selection



21

## Some tools for producing secure code

- Type safe languages (as opposed to C)
  - Buffer overflows are array bounds violations
  - Java, ML, ... check array bounds, prevent overflow
- Purify
  - Find memory errors on the heap
- Perl tainting
  - Track use of untrusted input
- Automated code analysis tools
  - Can catch many kinds of errors



22

## Purify

- Goal
  - Instrument a program to detect run-time memory errors (out-of-bounds, use-before-init) and memory leaks
- Technique
  - Works on relocatable object code
    - Link to modified malloc that provides tracking tables
  - Memory access errors: insert instruction sequence before each load and store instruction
  - Also uses a garbage collection technique to detect memory leaks



23

## Perl tainting

- Run-time checking of Perl code
  - Perl used for CGI scripts, security sensitive
  - Taint checking stops some potentially unsafe calls
- Tainted strings
  - User input and transitively values derived from user input
  - Result of matching against tainted string is not tainted
- Prohibited calls with tainted inputs
  - Print: this may write into a file
  - System: this can be used to execute arbitrary commands



24

## Safe Perl mail command (?)

- Check email string against pattern and parse

```
$email = $form_data{"email"};
if ( $email =~ /(w{1}[w-]*)@([w-.]+)/ ) {
    $email = "$1@$2";
} else { warn ("TAINTED DATA SENT BY ...");
    $email = ""; # successful match did not occur }
```
- What does this accomplish?
  - Only send email to address that "looks good"
  - Programmer responsible for "good" pattern
  - Perl cannot guarantee that email addr is OK



25

## Model Checkers

- Finite-State Verification Tool
  - System is modeled as a Finite-State Machine
  - Model Checker exhaustively tests each state
  - Correctness is defined as a set of boolean properties
  - Very good for detecting hard to find bugs (e.g. race conditions)
- Can model protocols and hardware effectively as FSMs
  - Example: John Mitchell's Analysis of SSL 3.0
  - Using Murϕ model checker
  - Problem: Software is not easily modeled as an FSM



26

## Automated code analysis for C

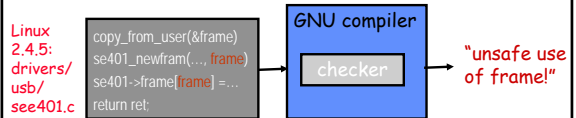
- Example tool
  - Ken Ashcraft and Dawson Engler, Using Programmer-Written Compiler Extensions to Catch Security Holes, IEEE Security and Privacy 2002
  - Used modified compiler to find over 100 security holes in Linux and BSD
  - <http://www.stanford.edu/~engler/>
- Benefit
  - Capture recommended practices, known to experts, in tool available to all



27

## Metacompilation (MC)

- Analyze compiler data structure to check code
  - Extensions dynamically linked into GNU gcc compiler
  - Applied down all paths in input program source
  - E.g., extension to check user input



Actual error in Linux: raid5 driver disables interrupts, and then if it fails to allocate buffer, happens with them disabled. This kernel deadlock is actually hidden by an immediate segmentation fault since the callers dereference the pointer without checking for NULL

## More Tools ...

- MOPS & BOON
  - Both from David Wagner & Berkeley
  - BOON looks at using integer logic to track buffer indices in programs. Attempts to identify potential buffer overflows
  - MOPS is a software model checker that attempts to verify programs for correctness
  - Links on the course webpage



29

## Conclusions

- Security takes extra effort
  - Know your security goals
  - Design with security in mind
    - Compartmentalize, least privilege
    - Minimize setuid, root
  - Implement carefully
    - Keep it simple
    - Think about attacks; secure the weakest link
  - Use tools that detect common coding problems
    - There are also tools that can analyze designs, but that's another story (harder to use, current research)



30

## Extra Slides

31

## Checking secure software

- Many rules for writing secure code
  - “sanitize user input before using it”
  - “check permissions before doing operation X”
- How to find errors?
  - Formal verification
    - + rigorous
    - costly, expensive. “Very” rare to do for software
  - Testing:
    - + simple, few false positives
    - requires running code: doesn’t scale & can be impractical
  - Manual inspection
    - + flexible
    - erratic & doesn’t scale well.
  - What to do??



32

## Example security holes

- Remote exploit, no checks

```
/* 2.4.9/drivers/isdn/act2000/capi.c:actcapi_dispatch */
isdn_ctrl cmd;
...
while ((skb = skb_dequeue(&card->rcvq))) {
    msg = skb->data;
    ...
    memcpy(cmd.parm.setup.phone,
           msg->msg.connect_ind.addr.num,
           msg->msg.connect_ind.addr.len - 1);
}
```



33

## Example security holes

- Missed lower-bound check:

```
/* 2.4.5/drivers/char/drm/i810_dma.c */
if(copy_from_user(&d, arg, sizeof(arg)))
    return -EFAULT;
if(d.idx > dma->buf_count)
    return -EINVAL;
buf = dma->buflist[d.idx];
Copy_from_user(buf_priv->virtual, d.address, d.used);
```



34

## User-pointer inference

- Problem: which are the user pointers?
  - Hard to determine by dataflow analysis
  - Easy to tell if kernel *believes* pointer is from user!
- Belief inference
  - “p” implies safe kernel pointer
  - “copyin(p)/copyout(p)” implies dangerous user ptr
  - Error: pointer p has both beliefs.
- Implementation: 2 pass checker
  - inter-procedural: compute all tainted pointers
  - local pass to check that they are not dereferenced



35

## Results for BSD and Linux

- All bugs released to implementers; most serious fixed

Violation	Linux		BSD	
	Bug Fixed	Bug Fixed	Bug Fixed	Bug Fixed
Gain control of system	18	15	3	3
Corrupt memory	43	17	2	2
Read arbitrary memory	19	14	7	7
Denial of service	17	5	0	0
Minor	28	1	0	0
Total	125	52	12	12



36

	Linux	BSD
Local bugs	109	12
Global bugs	16	0
Bugs from inferred ints	12	0
False positives	24	4
Number of checks	~3500	594



37

- ## 10 Platitudes
- [Viega and McGraw]
- Secure the weakest link
  - Practice defense in depth
  - Fail securely
  - Follow the principle of least privilege
  - Compartmentalize
  - Keep it simple
  - Promote privacy
  - Remember that hiding is hard
  - Be reluctant to trust
  - Use your community resources



38

- ## Secure the weakest link
- Think about possible attacks
    - How would someone try to attack this?
    - What would they want to accomplish?
  - Find weakest link(s)
    - Crypto library is probably pretty good
    - Is there a way to work around crypto?
      - Data stored in encrypted form; where is key stored?
  - Main point
    - Do security analysis of the whole system
    - Spend your time where it matters



39

- ## Defense in Depth / Fail securely
- Failure is unavoidable – plan for it
  - Have a series of defenses
    - If an error not caught by one, caught by another
  - Examples
    - Firewall + network intrusion detection
    - SSH + Tripwire
  - Clients, servers should not trust each other
    - Both can get hacked
  - Trusted code should not call untrusted code



40

- ## Principle of least privilege
- Give only the minimum privilege
    - Needed for the task, For minimum amount of time
  - Compartmentalize
    - Minimize damage possible from one module
  - Examples
    - Sendmail runs as root
      - Root privilege needed to bind port 25
      - No longer needed after port bind established
      - But most systems keep running as root
      - Also need root privileges to write to user mailboxes
    - Will look at qmail for better security design



41

- ## Keep It Simple
- Use standard, tested components
    - Don't implement your own cryptography
  - Don't add unnecessary features
    - Extra functionality => more ways to attack
  - Use simple algorithms that are easy to verify
    - A trick that may save a few instructions may
      - Make it harder to get the code right
      - Make it harder to modify and maintain code



42

## Promote Privacy

- Discard information when no longer needed
  - No one can attack system to get information
- Examples
  - Don't keep log of old session keys
  - Delete firewall logs
  - Don't run unnecessary services (fingerd)



43

## Remember that hiding secrets is hard

- Information in compiled binaries can be found
- Insider attacks are common
  - Companies spend time/money on firewalls
  - Firewalls do not protect against inside attack
- Security by obscurity doesn't work!!!



44

## Use Your Community Resources

- Consult experts
- Allow public review
- Use software, designs that other have used



45

## Secure Implementation

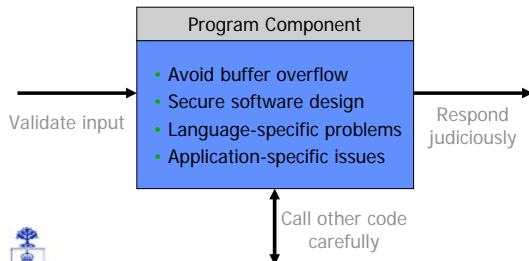
- Generalities apply to design, implementation
  - Example: defense in depth within code
    - Check privileges several times
    - Helps if code is modified
    - Also can be useful if there is a race condition ...
- Once you have a design, implement carefully
  - Keep security issues in mind
  - Some tools can help



46

## General categories

[Wheeler]

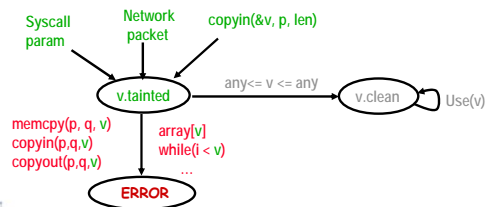


See Wheeler's book on web

47

## Sanitize integers before use

Warn when unchecked integers from **untrusted sources** reach **trusting sinks**



Linux: 125 errors, 24 false; BSD: 12 errors, 4 false

48