CNS Lecture 14

Writing secure code
Crypto APIs
Secure applications

594 paper due 12/1/06
Assignment 10 forensics

In the news

• Remember we said to avoid cross-site scripting use HTMLEntities() in PHP -- well today, buffer overflows in this function
• In case you hadn’t noticed, about 9 out of 10 emails today are spam, sent by botnets around the world
• GNU Radius format string vulnerability
• Acer Notebooks ActiveX arbitrary command execution

You are here …

Attacks & Defenses
• Risk assessment
• Viruses
• Unix security
• authentication
• Network security
Firewalls, sys/Sec/DS
• Forensics
• Secure coding

Cryptography
• Random numbers
• Hash functions
• MD5, SHA, RipeMD
• Classical + stege
• Number theory
• Symmetric key
• DES, Rijndael, AES
• Public key
RSA, DSA, D-H, ECD

Applied crypto
• S/SH
• PGP
• S/Mime
• SSL
• Kerberos
• IPsec
• Crypto APIs

Software security

• Software engineering
  – Why do smart people write buggy software?
• Writing secure code
  – Secure design
  – Secure coding & tools
  – Testing for security
  – Defensive programming

First computer bug?

Software bugs

• Flaw in software are expensive -- $60 billion/yr (NIST)
• Well known bugs
  – Denver baggage handling system – complex (500 computers), late and 50% cost overrun, abandoned
  – Ariane 5 (1996) – loss of guidance system, fit. pt. to 16-its int. overflow
  – USS Yorktown (1998) – aircraft carrier dead in the water for hours due to zero divide
  – Mars Climate Orbiter (September 23rd, 1999) – $125m loss due to failure to convert meters to feet
• Just examples of when things go bad on their own. What if you have an active adversary who is looking for and exploiting bugs? Making an "unlikely event" a certainty in the millions of copies of the software in use around the world.
Bug density?

- 10 bugs per 1,000 lines of code?
  - Million lines of code = 10,000 bugs
- Complexity breeds bugs
- Of course all of these bugs aren’t exploitable
- But it’s a bug hunter’s paradise
  - The bad guys are winning

Why is software development so hard?

- Complex
  - No two parts the same
  - Growth adds to complexity in non-linear fashion
  - Interaction between data, function, user, devices
- Conformance to human requirements
- Changeable
  - Malicious
  - Easy to change, as let’s change it
- Structure is impossible to visualize
- Human intensive
- Market pressures
  - Eliminate all bugs, never ship
  - Ship now, let customers debug

Risk Exposure

\[
RE = P(L) \times S(L)
\]
- Risk Exposure: \( RE = \text{Prob}(\text{Loss}) \times \text{Size}(\text{Loss}) \)
  - Loss due to unacceptable dependability
  - Loss due to market share erosion

Internet Startup

\[
RE = P(L) \times S(L)
\]
- Loss due to unacceptable dependability
  - Loss due to market share erosion

Safety-Critical or Secure System

\[
RE = P(L) \times S(L)
\]
- Loss due to unacceptable dependability
  - Loss due to market share erosion
Dilemma

Cost

Schedule

Quality/Security

Pick any two.

Security conflicts with
• Ease of use
• Speed
• Feature richness

Software development

• You hack up some code
  - Single developer
  - "Toy" applications
  - Short lifespan
  - Single or few stakeholders
  - Architect = Developer = Manager = Tester = Customer = User
  - One-of-a-kind systems
  - Built from scratch
  - Minimal maintenance

• Enterprise strength
  - Teams of developers with multiple roles
  - Complex systems
  - Indefinite lifespan
  - Numerous stakeholders
  - Architect ≠ Developer ≠ Manager ≠ Tester ≠ Customer ≠ User
  - System families
  - Reuse to amortize costs
  - Maintenance accounts for over 60% of overall development costs

Software development lifecycle

Waterfall Model

Requirements

Design

Implementation

Integration

Validation

Deployment

Lots of methodologies

clean room

PSP

SDM

pair programming

agile

object-oriented, reuse

Playing the software development game ...

• The trusting model
  - Design/code/test to provide proper function
  - Bugs are random and rare

• The defensive model
  - Design/code knowing you have an active adversary
  - Test not only that the right things happen, but also that wrong things don’t happen
  - Bugs may be vulnerabilities and exploited

Computers at Risk

“The developers of secure software cannot adopt the various probabilistic measures of quality that developers of other software can. For many applications, it is quite reasonable to tolerate a flaw that is rarely exposed and to assume that its having occurred once does not increase the likelihood that it will occur again. It is also reasonable to assume that logically independent failures will be statistically independent and not happen in concert. In contrast, a security vulnerability, once discovered, will be rapidly disseminated among a community of attackers and can be expected to be exploited on a regular basis until it is fixed.”

Software engineering for security

• Secure design
• Secure implementation
• Security testing
• Secure deployment
• Root cause analysis for bugs
• Policy and training
Secure design

- Assess the risk
  - Detail your assets
  - Know the threats and your attackers
  - Mitigate the threats
  - Cost: time & money ... acceptable risk
- Formal threat modeling
  - Attack trees
  - Reduce the attack surface
- Apply secure design principles
- Think like a bad guy

Design principles

- Principle of least privilege
  - Give only those privileges needed to complete a task
- Principle of fail-safe defaults
  - Access should be denied unless it is specifically permitted
- Principle of economy of mechanism
  - Security mechanisms should be as simple as possible
- Principle of complete mediation
  - All accesses to objects must be mediated
- Principle of open design
  - Security should not depend on secrecy of design or implementation
- Principle of separation of privilege
  - Don’t grant permission based on a single condition (su: password+wheel grp)
- Principle of least common mechanism
  - Mechanisms used to access resources should not be shared
- Principle of psychological acceptability
  - Security mechanisms should not make resource access more difficult

Secure implementation

- Trained developers
- Language choice: Java, C++, C#, VB, perl, php, ...
- Trusted API’s
- Development tools
  - Source code control
  - Compile/link/debug
  - Compiler warnings
- Peer code review
  - Specifically “security reviews”
  - Use checklists
- Static analysis
  - Microsoft PREFER and PREFast
  - RATB, ITSO4, Klocwork, Covertly, ESC/Java

Code reviews

- Probably your best return on investment for security
- Will increase coding “costs”, but worth it for production software
- Identify critical components for review
  - Processes that run with elevated privilege
  - Processes that talk to the net
  - Software that parses packets/input
  - Use checklists of things to look for:
    - Use of tainted data, banned functions, false assumptions
    - Failure to check return code, signed/unsigned, toctou
    - Update the list as bugs are found
- Sign off on review
- Engineers need to be trained to recognize flaws/vulnerabilities
- What about legacy code? Or outside libraries (e.g. OpenSSL)

Threat Modeling Process
Identify the Threats by Using Attack Trees

How are the bad guys finding bugs?

- Open source → code review
  - grep for str*()
  - Analyze input and packet parsing
- Blackbox testing
  - “random” alteration of input — big username, missing fields ...
  - Caused a crash
  - Analyze dump
  - Disassembly (IDA Pro)
- Cause a crash
  - Analyze dump
  - Experiment
- Bug announcements, or diff patched vs old images
  - Attack unpatched versions
- Motivation?
  - Attribution
  - Financial gain
Sample code

```c
/* really bad telephone number lookup code, expects "command" on stdin */
static char cmd[256];
static char format[] = "grep %s phone.list\n";
main(){
    char buff[256];
    gets(buff);
    sprintf(cmd,format,buff+5);
    printf(cmd);  // debug, remove later
    system(cmd);
}
```

More broken code snippets

```c
char *p = NULL;
if (argc == 2)  p = argv[1];   // suggest if (2 == argc) ...
    why "p = "?

while(i<6){
    int x = 5;
    p = malloc(1024);
    return (i+5);
}
```

Static analysis

- Automated source code analysis
- Looking for:
  - Potential NULL pointer dereferences
  - Access to an unknown area (e.g., array or dynamically allocated buffer)
  - Write to an object that has already been deallocated
  - Use of memory that has already been deallocated
  - Use after free
  - Out of bounds
  - Save/return values (e.g., returning the address of an automatic variable from a subroutine)
  - Failure to set a return value from a subroutine
  - Buffer and array underflows

- List of warnings, many false positives
- 10x slower than compile

Coverity's static analysis of Linux kernel

- Growth in Linux kernel in drivers
- Complexity grows but not % of security flaws
  - 2001 2.4.1 kernel 1.6 MLOC 1000 flaws
  - 2004 2.6.4 kernel 5 MLOC 900 flaws

Coverity's analysis of open source software

- Less than 1 bug per 1000 lines of code

Testing for security

- Regression testing, unit testing, integration testing, system testing
- Most testing is aimed at testing conformance/quality, that code does what it's supposed to do
- Security testing is different, making sure bad things don’t happen
  - If you know to test for it, you wouldn’t have coded it wrong...
  - Independent test group
Secure testing

- **Run time tests**
  - Memory leak tester (purify)
  - Fuzz testing, random user/packet input (udpsic)
  - Smarter fuzz testing -- codenomicon
  - Lots of web app testing products

- **Penetration testing**
  - Think/attack like a bad guy
  - Monitor bugtraq/CERT, because the bad guys are pen testing!

Software Defect Detection Opportunity Tree

Defense in Depth

- **Training, code review, testing won’t find all the bugs (prevention)**
- **Buffer overflow defenses**
  - Stackguard/propolice
  - Hardware memory protection (RO/NX)
  - Address randomization (text/heap/stack/data) ... see PaX
- **Fail gracefully, fault tolerance**
- **Log events and error conditions**
- **Watch for crashing daemons/servers**

Security training

- **Educate your designers, developers, testers, and users**
- **How can you do a risk assessment, if you don’t know the risks?**
- **How can you do code reviews looking for security flaws, if you don’t know what security flaws look like?**
- **Yearly re-train?**
- **Metrics? Bugs/1000 lines of code**

Factor-of-100 Growth in Software Cost-to-Fix

- **Root cause analysis**
  - Each defect found can trigger five analyses:
    - Debugging: eliminating the defect
    - Regression: ensuring that the fix doesn’t create new defects
    - Similarity looking for similar defects elsewhere
    - Insertion: catching similar defects earlier
    - Prevention: finding ways to avoid such defects
- **Example, strcpy() cause for buffer overflows**
  - ban str*() and provide safe string libs
  - Training
  - Static analysis tools to detect
Software Defect Reduction Top-10 List

1. Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.
2. About 40-50% of the effort on current software projects is spent on avoidable rework.
3. About 80% of the avoidable rework comes from 20% of the defects.
4. About 80% of the defects come from 20% of the modules and about half the modules are defect free.
5. About 90% of the downtime comes from at most 10% of the defects.
6. Peer reviews catch 60% of the defects.
7. Perspective-based reviews catch 35% more defects than non-directed reviews.
8. Disciplined personal practices can reduce defect introduction rates by up to 75%.
9. All other things being equal, it costs 50% more per source instruction to develop high-dependability software products than to develop low-dependability software products. However, the investment is more than worth it if significant operations and maintenance costs are involved.
10. About 40-50% of user programs have nontrivial defects.

Microsoft secure coding

- Security push – audit legacy code for security flaws
- For Vista, Security Development Life Cycle
  - Design – threat modeling, load, privilege, review
  - Implementation – banned APIs, tools, review, bluehat
  - Deployment – secure by default, root cause analysis of bugs
- Security training – from one course to a dozen
- Address Space Layout Randomization (ASLR)
  - Randomize text/data/heap
  - Stackguard (/GS)
- Support RO/NX
- TPMs and Bitlocker
- Upper management support

Secure Product Development Timeline

1. Analyze threats
2. Secure product development
3. Determine security sign-off criteria
4. Test for security vulnerabilities
5. Learn and refine

Design security in from the beginning

- Good design and code reviews are more effective than testing
- Should the software vendor be held accountable for loss due to software flaws?

Adding security in

- Our ncp labs (assignments 4, 7, 6)
  - Design problems
  - Coding problems
- TCP protocol
  - Assumed “cooperating” processes
  - SYN flooding, malformed packets...
- NTP protocol
- SNMP protocol (Simple Network Management Protocol)
  - Or: Security/Not My Problem
  - Clear text/resolved group (v1)
  - Added notion of view (ACLs/least privilege) (v2c)
  - Added authenticity/privacy (v3)

SNMP

Simple Network Management Protocol
- Used to manage network devices (routers, switches, toasters,...)
- Network manager and database (MIB) and network agents
  - ASN.1/BER data encoding
  - Object ID (OID) (long numeric tag) provides unique variable names (tree)
- Simple datagram protocol (UDP port 161)
  - GET GET NEXT GET RESPONSE GET TRAP (port 162)
- Version 1, simple (no) security (community string)
  - SET's disabled
  - Security for SNMP v1: Security Not My Problem
- Numerous implementation flaws discovered with Finland/Codenomicon smart fuzz tester
  - Buffer overflows community string
  - ASN.1 BER bugs (Tag Length Value TLV)
SNMP v3 security (RFC 3414)

**Threats**
- Impersonation
- Modification
- Replay/re-order/delay
- Disclosure

**Countermeasures (v3)**
- Timezone
  - Loosely synchronized monotonically increasing time indicators
- Authentication
  - MD5 HMAC (96 bit)
  - 128 bit (16-byte) key
  - SHA optional (crypto agile)
- Privacy
  - DES/CBC (AES & D-H option)
  - 8-byte key & 8-byte IV (from key)
- Export restrictions
- Additional control thru router filters

Example CAPIs

- OpenSSL or Young’s ndes/SSLEasy or RSA’s BSAFE
- RSA/DES/MD5, SHA
- Cryptix API
- Certificate mgt (keytool)
- Public key (RSA/DSA), SSL, and D-H
- Hashes, block ciphers, PRNG
- Provider-based architecture (CSPs)
- Exportable
- Shareware packages
- OpenSSL API

Java JCE

- Exportable
- Provider-based architecture (CSFs)
- Hashes, block ciphers, PRNG
- Public key (RSA/DSS), SSL, and D-H
- Certificate mgmt (keytool)
- Signing (CA)
- Signing JARs
- Also Cryptix API

Crypto APIs

- Functions for developing crypto applications
- Provide data privacy and integrity and authenticity
- Portable -- architecture and OS
- Services
  - Establish context (keying, algorithm negotiation)
  - Encrypt/decrypt and authenticate
  - Certificate mgmt (fetch, URL)
  - Sign/verify
  - Encode/decode (PKCS, ASN)
  - Protocols (LDAP, SSL/TLS, key update)
- Multiple algorithms -- symmetric, asymmetric, hash, random, primality, big numbers, compress, encode

OpenSSL API

- Widely used reference implementation
- Good basic crypto tools
- Hashing (MD4, SHA, PK14)
- Random numbers and prime numbers
- Big number library
- Symmetric key crypto (DES, Blowfish, AES, RSA)
- Public key crypto (RSA, DSS, D-H, ECC)
- Support for crypto hardware accelerators (engines)
- Cryptix agile wrapper (EVP)
- SSL/TLS
  - API for creating SSL network connection (accept, sockfd)
  - Commands for key/certificate management (your own CA)
  - New -- key generation
  - Verify -- verify.c
  - Public --- (server cert encode)
  - CRL -- ca's mgmt
- Keep current
  - New features
  - Bug fixes

CNS Lecture 14 - 47
CNS Lecture 14

CNS Lecture 14

Kerberos
- Client context established (tickets, keys) KDC
- Server/Principal must be registered
- krb_mk_safe(), krb_mk_priv()
- user provides transport

DCE
- context established (tickets, keys) KDC
- Server/Principal must be registered
- rpc_binding_set_auth_info()
- RFC mechanism provides transport

Kerberos v4 client

main()
|
| EVPNT_EXIT k;  /* Kerberos data */
| EVPNT_EX m;  /* Get ticket for server from TGS, create krb_mk_req message */
| EVPNT_EXIT k;  /* Get Kerberos realm of host */
| EVPNT_EXIT k;  /* Get ticket for server from TGS, create krb_mk_req message */
| EVPNT_EXIT k;  /* Get Kerberos realm of host */
| EVPNT_EXIT k;  /* Get ticket for server from TGS, create krb_mk_req message */
| EVPNT_EXIT k;  /* Get Kerberos realm of host */
| EVPNT_EXIT k;  /* Get ticket for server from TGS, create krb_mk_req message */
| EVPNT_EXIT k;  /* Get Kerberos realm of host */

Kerberos v4 server

Main()
|
| EVPNT_EXIT k;  /* Kerberos data */
| EVPNT_EXIT k;  /* Check Kerberos context */
| EVPNT_EXIT k;  /* Check Kerberos context */
| EVPNT_EXIT k;  /* Check Kerberos context */
| EVPNT_EXIT k;  /* Check Kerberos context */
| EVPNT_EXIT k;  /* Check Kerberos context */
| EVPNT_EXIT k;  /* Check Kerberos context */
| EVPNT_EXIT k;  /* Check Kerberos context */
| EVPNT_EXIT k;  /* Check Kerberos context */
| EVPNT_EXIT k;  /* Check Kerberos context */

Kerberos v4 API example
- client sends cleartext and private message
- server decodes and prints
- must be registered with Kerberos and server key in server's /etc/krb5.conf

krb_mk_safe(authent, service, instance, realm, checksum)
krb_mk_req(authent, service, instance, realm, credentials)
krb_priveq(in, out, in_length, schedule, key, sender, receiver, msg_data)
krb_priveq(in, out, in_length, key, sender, receiver, msg_data)
krb_priveq(in, out, in_length, key, sender, receiver, msg_data)
Server cont.

```c
/* GET KRB_MK_SAFE MESSAGE */
i = sizeof(c_sock);
i = recvfrom(sock, (char *)ktxt->dat, MAX_KTXT_LEN, flags, 
             (struct sockaddr *)&c_sock, &i);
/* Verify the checksummed message */
if (i != KSUCCESS) {
    // Handle failure
    printf("Safe message is: %s\n", msg_data.app_data);
    return -1;
}
```

```
printf("Decrypted message is: %s\n", msg_data.app_data);
```

kerberizing

- you can add Kerberos calls to your own client/servers
- need Kerberos data base, authenticator, ticket-granting server, and administrative programs
- can use klogin, but better if you have kerberized BSD utilities
- Kerberos calls added to login, r-utilities, NFS

GSS

Generic Security Service

- IETF RFC 1508, 1509, API: RFC 2743, 2744
- establishes a security context and provides security services
- application goes through 4 phases
  1. establish identity (authentication)
  2. negotiate shared security context
  3. exchange messages - privacy, integrity
  4. destroy context
- sample implementations in Kerberos v5 and DCE distributions
- API is independent of OS and network protocols
- application is responsible for message transport or file I/O
- used in global (grid security)
- Java class GSSUtil
- primary functions
  - gss_acquire_cred()  
  - gss_init_sec_context() 
  - gss_seal()        gss_unseal() 
  - gss_sign()        gss_verify() 

Example CAPI

Microsoft CAPI

- 23 crypto services
- context and key generation
- key exchange
- encryption/signing
  - CryptAcquireContext()  CryptGenKey()  CryptGetKey()
  - CryptEncrypt()  CryptDecrypt()  CryptHashData()
  - CryptSign()  CryptVerify()  CryptGetRandom()

Other proprietary CAPIs - Sypnis or Cyrix

Where to encrypt?

- link layer
  - encrypting modem, net board (wireless)
  - transparent, fast
  - protects only one link (pt-to-pt)
  - info may be exposed in OS
- network/transport layer
  - IPsec, IPv6(v4), VPN
  - transparent
  - selectable (policy)
  - apply/host/net keying
- application layer
  - custom applications (TCP, etc)
  - CAPIs can help (openSSL)
  - intrusive, but flexible
  - key for every logical circuit
Secure applications

Apps we’ve already seen

- ssh
- PGP
- SSL/NetScape
- S/MIME

Characteristics

- Key management / schedule
- Cipher / hash functions
- Public key / symmetric key
- Random numbers
- Protocols and encodings
- Crypto agile

Nautilus

Nautilus sinks clipper ships ...

- Encrypted Internet / modem phone
- UNIX, Windows
- Various audio encodings
- D-H or shared secret
- 3DES, Blowfish, IDEA
- Random bits from microphone noise

Speakfreely

- Use microphone / speaker for secure voice over IP
- Uses GSM compression, SIGA AtoD PDM, VAT / RTP options
- Crypto
- AES / Blowfish / IDEA / DES to encrypt audio (CBC within a "block")
- Session key can be provided by user as ASCII passphrase or by a key file (e.g., dd count=10 of= rand.dat if=/dev/random)
- sfmike will generate a session key (128 bit encoded into 32 ASCII characters that Bob could PGP email to Alice, or read over phone?)
- For “conference call” -Z option will create session keys for each user and PGP / GPG encrypt the key with their public key

Zfone

- Zimmerman’s voice-over-IP secure phone (IETF draft)
- Uses ideas from PGPfone
- Doesn’t need PKI
- Establishes ephemeral session keys in for Secure RTP
- Displays authentication string for each user to verify authenticity and prevent man-in-the-middle attacks
- Can include pre-shared secret and running shared secret
- Secure RTP uses
  - HMAC-SHA1
  - AES / Blowfish
  - 128-bit session / key
  - 256-bit key derivation rate

Disk encryption cfs

- Uses passphrase encrypted directories
- Blaze AT&T
- Most BSD-based UNIX and Linux
- Local NFS server on loopback interface
- Encrypted directories
- Encrypted directories can be normal
- /crypt is mounted
- Command to attach / detach (with DES key)
- If CFS unavailable, crypt / dev can be used

Startup cfs

- mount -o port=3049,intr=localhost:/null /crypt
- cattach /usr/dunigan/secrets tom
- Key: xxxxxxxxx
- Do normal file operations cat, vi, etc.
- cdetach tom
- Check /usr/dunigan/secrets/80b6e856778
- Contents are gibberish (encrypted)

CFS encryption

- Need random access for file
- Non-cryptic block update
- CFS computes two DES keys from passphrase
- Uses first key to create DES block-random-bit mask using DES-CFB
- Stores DES-CFB block in block of mask (512 bytes)
- Optionally, mask is generated from inode number and stored in file system field
- 4 times slower than regular file I/O benchmarks, but only about 1.3 times slower for "normal" applications (compared)

Are plaintext versions lying around? Swap space, backup files (vi), /dev/mem...
- Why not to encrypt at all or whenever?...
Windows Encrypting Files System (EFS)

- Encrypt folders (DES-X or 3DES)
- Provides key recovery; you need a certificate

Windows EFS

Windows generates a random number for the file encrypting key (FEK). User’s public key is used to encrypt the FEK. Encrypted key is stored in extension attribute to the file. FEK also encrypted under recovery agent’s public key.

“wipe” service provided to scrub clear text file blocks.

Bitlocker vista

- Drive encryption optionally using TPM, optional user password
- Solve the stolen laptop problem
  - Removing the drive or boot with thumb drive and becoming Administrator will doesn’t give you access to the Bitlocked disk
- Each sector encrypted with AES CBC
  - no authentication? Because it would require more bits per sector
  - Direct (CBC) any change will affect the rest of the sector
- User “Elephant diffuser”
  - Provides poor man’s authentication
  - Fast (32-bit word XOR and rotate)
- Performance (pentium 4)
  - AES 20 cycles/byte
  - Diffuser 10 cycles/byte
  - 5% slower

MAC disk encryption

- Encrypt a disk “volume” using AES
  - Key can be added to your keychain for one-time signon
- FileVault to AES-encrypt your home directory
  - Key recovery option with Master password
  - Disable auto-login (dunno)
  - Backups unencrypted (unless you backup the vault)

Next time

Review