COMS E6998-9: Software Security and Exploitation

Lecture 7: Vulnerabilities Cont. + Data Security and Cryptography

Hugh Thompson, Ph.D.
hthompson@cs.columbia.edu

© Hugh Thompson 2010
More Vulnerabilities
XSS

• Cross site scripting (XSS) vulnerabilities come from user data being displayed on web pages
• If that data is malicious, attackers can gain access to account information, data, and functionality accessible by the victim
• Can be one of the easiest attacks to carry out
• Fortunately there are many anti-XSS libraries available (depending on your platform) that you can use to sanitize data
Command Injection

• Command injection is not limited to SQL or JavaScript
• Any time user data is mixed with executable code there is the potential of command injection
• You should never mix code with data given the potential for manipulation
Beware 2\textsuperscript{nd} Order Attacks

• 2\textsuperscript{nd} order attacks occur when malicious data passes through initial defenses, is stored, and then accessed later by a vulnerable function

• Particularly dangerous because they are almost always missed by security testing

• The threat of 2\textsuperscript{nd} order attacks builds a strong case for properly sanitizing data that is to be stored
Canonicalization Problems

• Canonicalization is the process of converting data that has multiple representations to a single standard “canonical” form

• The problem is that there are alternate ways to represent things:
  – Files, URLs, IP addresses, Paths, etc.

• Sometimes there are strange representations you don’t expect...
Canonicalization Problems

• Windows example:
  “securecoding.txt” ≠ “secure~1.txt”

• BUT...the OS can treat all of the following as the same:
  – securecoding.txt
  – secure~1.txt
  – securecoding.txt::$DATA
  – securecoding.txt.
  – Many more!

• Attackers can leverage this fact to create inputs that sneak past filters yet force the application to access an unintended resource
Unicode Threats

• When dealing with Unicode strings, string comparison failure can cause unexpected results

• Always consider:
  – When evaluating size, what’s being measured
  – Some string comparisons are byte-wise
Data Security and Cryptography
Developer’s Cryptography Primer

• Cryptography is good at protecting data at rest or in transit. It does little for data in use and protects against only a very narrow set of threats.

• Symmetric Cryptography - a single shared secret is used to encrypt and decrypt.
  Methods: 3DES, AES, etc.

• Asymmetric Cryptography – a different secret is used to encrypt and decrypt.
  Methods: RSA, etc.
## Symmetric vs. Asymmetric Cryptography

<table>
<thead>
<tr>
<th>Symmetric Key</th>
<th>Asymmetric Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires a shared secret to be maintained</td>
<td>There are two keys used</td>
</tr>
<tr>
<td>Computationally cheap</td>
<td>Computationally expensive</td>
</tr>
<tr>
<td>Difficulties in transporting the key securely (key exchange problems)</td>
<td>Public Key Infrastructure (PKI) can be used to distribute keys via a trusted 3rd party</td>
</tr>
<tr>
<td>Used widely because of computational ease</td>
<td>Typically used to verify authenticity (through an encrypted hash) or to exchange a private key</td>
</tr>
</tbody>
</table>
On the Replace-ability of Algorithms

- As computational power increases, the effectiveness of some cryptographic implementations decreases.
- Many companies have switched from DES to 3DES to AES.
- This speaks to the fact that we need to write code with the foresight that the cryptographic implementation used may need to change.
- One of the biggest reasons that encryption breaks down is through improper implementation: don’t code your own implementation or create your own algorithm.
Practical key management

• Poor key management is one of the biggest weaknesses in a cryptographic implementation

• Top things to remember:
  – A key stored in source code can be found through simple reverse engineering
  – A key stored in a configuration file can be found even more simply
  – A key stored in your application can be replaced (if the attacker has access to the application)
Hashes

• A cryptographic hash function is a reproducible method of turning a large message into a small digest (hash) of that message
• Hashes are good to help verify the integrity of data
• It should have the following properties:
  – One-way: It should be hard to find a message that yields a given digest (basically we shouldn’t be able to reconstruct the message from the digest)
  – No collisions: It should be difficult to find two messages that produce the same hash
  – No iterative refinement: A small change in the message should result in an unpredictable change in the hash
Hashes

• Popular cryptographic hashing algorithms are MD5 and SHA1

  This text is going to change slightly
  SHA1 Hash: 2ea1565ca629e2c1562d9a7aa5d1b15dbc2604e7
  MD5 Hash: 700a21e5bceef03b1ada0b59bf98705c

  This text is going to change slightly
  SHA1 Hash: 5e6a81193372ac6b2450d3a17156079c264dfbea
  MD5 Hash: b594837fd1c1dd53c2db5c34ee52b7c5

• Hashes are often encrypted using asymmetric cryptography to create digital signatures that can verify a message’s authenticity and integrity
Data Integrity

• Data integrity means that data has not been tampered with and is complete
• A common means of ensuring data integrity is to use digital signatures (encrypted hash values)
• Remember that if you store a hash or a key in your code that it can be both reverse engineered out and replaced
Memory protection - Dangers

• Secrets left in memory are risky for the following reasons:
  – A crash could cause memory data to be dumped to disk
  – Data may remain in memory long after it is addressable
  – Memory locations can be reclaimed by the JVM and thus be used in ways never expected (such as to pad packets)
Memory protection – Solutions

• Overwrite sensitive data before freeing or returning from a function
• Consider in-memory encryption
  – (CryptProtectMemory() in Windows for example)
• Lock sensitive data so that it cannot be paged to disk (mlock(), mlockall(), and VirtualLock() )
  – Warning: This can have severe performance impact
Random number generation

• Don’t use RAND – it has poor entropy and predictability
• Use a cryptographically secure pseudo-random number generator (CSPRNG)
• These are available in crypto libraries such as OpenSSL
• Consider the consequences of a user being able to predict random number values:
  – Cookies; encryption keys; filenames; behavior; etc.
Introduction to secure audit and log

• Tips on Logs:
  – Should only be writable by administrator/root and the application that is logging
  – Should be considered highly sensitive
  – You should ensure that attackers cannot fill logs by checking on length and taking standard precautions (such as a new log file every day)
  – Remember that sometimes regulations dictate some information that must be logged and some information that cannot be logged
  – Failed password attempts can hold sensitive data!
  – Consider using hashed if the only purpose of a piece of information is confirmation
Reverse Engineering and Avoiding Security by Obscurity

• You must assume that if users have your binaries then they have your source code
• The compiler is not an encryption tool
  – Any encryption keys can be easily located using an entropy scanner
• Tools such as IDA Pro, JAD, etc. make decompilation incredibly simple
• Commercial solutions to obfuscation exist but they serve to further delay reverse engineering, not prevent it (code eventually must be decrypted to run)
Security Testing Part 1: Fuzzing
Introduction to Fuzzing

• What is fuzz testing (or “fuzzing”)? Answer: Data corruption, plain and simple
• Fuzzing catches the cases you wouldn’t think about...weird input, corner and fringe cases, cases that you may not expect to be bug revealing
• Tend to find: Buffer Overflows, Format String issues, traversal problems, and many others --- it all depends on the symptoms you look for
• Great because its automated
• A few things to define up front
  – Which interfaces does your application have?
  – What protocol/format/etc. is used?
  – What to define as a symptom of failure? (needed for automation --- more on this later)
Benefits of Fuzzing

- Highly automatable
- Finds vulnerabilities that wouldn’t normally be detected during functional testing
- Inexpensive...free tools, low cost tools, or grow your own
- Can unearth some of the most critical vulnerabilities in software
- Gives a good sense of how robust an application is

© Hugh Thompson 2010
What can be fuzzed?

• Anything that accepts input:
  – APIs
  – File parsers
  – Network protocols
  – GUIs
  – CLIs
  – Web services calls
  – URLs
  – Script
Data Corruption

- Long strings
- Special characters (general strings)
  - %s, /n, ‘, AUX, “../..”, ...
  - Canonicalized forms of these can also be cool!
- Special values (numeric types)
  - 127, 0, -128, 32756, ...
  - Boundary values and boundary values + 1 have been particularly interesting
- Other variable types and interesting strings
  - domain names, company names, ...
- Other techniques: Flipping bits, swapping data, removing delimiters, incrementing values, ...
Delivering the data to the application

- Fuzzers need to create a container to deliver data
- May have to do things like:
  - Recalculate checksums
  - Add special identifiers
  - Mark certain sections to NOT corrupt (header tags, etc)
  - Encode your corrupted data (Base64, etc)
  - Encrypt your data (best in this case to corrupt BEFORE data hits encryption routines ((unless you’re testing the parser)))
  - Create/Maintain containers (SOAP, HTTP, etc.)
  - Maintain type (for API parameters for example)
Types of Fuzzing

• Random (dumb)
  – Inserts random characters and strings
  – Not aware of much ... may require a wrapper to be effective
    (eg SOAP, TCP/IP, file header, )

• Context-Aware
  – Has some notion of format and type (XML, APIs, etc.)
  – May use a bounded set of “random” data (integers, script tags, ...)

• Adaptive
  – Changes the next corruption data based on execution
  – May be used to “cover” more of the application
Illustrating the difference...Random

Pure random fuzzing may only test initial routines and not pass a formatting / protocol check by the application.
Illustrating the difference...Context-Aware

• Context-Aware testing may be necessary to get deeper into the application

• May need to bypass things like checksums, etc.

• May also included wrappers for protocols, formats, SOAP, web services, RPC, ..., you name it!

• Likely to get more “coverage” of the application binary
Illustrating the difference... Adaptive

- Adaptive fuzzing uses feedback from the application or the debugger to modify the data that’s sent.
- Can be used to get in-depth coverage on a binary
- Will adapt the data to get through branching “gates”
- Can find some really interesting vulnerabilities... especially combined with a binary scanner (like the BugScam IDC package)
A Word on Oracles

<table>
<thead>
<tr>
<th><strong>or·a·cle</strong> - Function: <strong>noun</strong> Etymology: <em>from Latin oraculum, from orare to speak</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a</strong> : person giving wise or authoritative decisions or opinions</td>
</tr>
<tr>
<td><strong>b</strong> : an authoritative or wise expression or answer</td>
</tr>
</tbody>
</table>

- In testing, an “oracle” is something that gives you the right answer.
- For fuzzing, we can only identify problems if an output is different than what we expected and we can compare the two (i.e. given by the oracle).
- For example, how would you automate the testing of a scientific calculator?
Things to look for (oracle)

• (EASY) Exceptions (first or second chances, read/write AV, ...)
  – Tools: Debugging APIs

• (MEDIUM) Resource consumption (memory, disk, network, CPU)
  – Tools: Debugging APIs, Resource Monitors, Memory leak detectors

• (MEDIUM) Sandbox escape (folders, IP address ranges, APIs, registry, ...)
  – Tools: Holodeck/HEAT APIs, Detours, FileMon, RegMon, ...

• (ADVANCED) Binary/Data coverage or pattern changes
Stopping Criteria

• The decision that you’ve done “enough” fuzzing is really based on internal quality tolerance --- BUT --- in many cases it makes sense to continue fuzzing even after product release.

• There aren’t great answers here but here are some criteria:
  – Business tells you to ship the product - even then it may make sense to fuzz till EOL
  – Binary coverage (we’ve touched x% of the reachable code through fuzzed inputs)
  – No differential coverage (only already-traversed paths get explored)
  – X number of “clean” runs

© Hugh Thompson 2010
Challenges / Future of Fuzzing

- Metrics!
- Distributed fuzzing (divide and conquer)
- “Oraclings” agents on server (used by some folks now)
- Good tools designed for fuzzing in a broader environment
- Knowledge sharing on fuzzing
Summary

• Fuzzing is interesting but it should only be part of an overall security strategy

• If there is poor coverage as you fuzz, it's likely that the application may be rejecting large chunks of data because it’s malformed – fuzz more granularly

• Fuzzing has already made its way into web application testing tools – expect broader tool support to exist