Software Security Course

Parsing and other file-related bugs

Dimitrios A. Glynos
{ daglyn at unipi.gr }

Department of Informatics
University of Piraeus

Part I

Introduction

File-related bugs

- They are ubiquitous
- They can be critical (may lead to privilege escalation, remote code execution etc.)

Major categories of file-related bugs

- File handling
    - incorrect file permissions
    - insecure file open
    - ..
- File writing
    - exposure of sensitive information from uninitialized buffers
    - depletion of storage resources
- File parsing
    - insecure copy of data structures found in files
    - arbitrary content injection attacks
    - insecure deserialization
- We have examined some of these in previous lectures
- A more focused look will be provided in this lecture

Part II

File handling bugs

Incorrect file permissions

- CVE-2005-2962: `ntlmaps` is an NTLM authentication proxy server; the post-installation script of `ntlmaps` changes the permissions of the `ntlmaps` configuration file to be world-readable.
- This configuration file typically contains the administrative username and password of the Windows NT system that is used as the NTLM authentication server, thus leaking these credentials to local users.
- Does not follow the *principle of least privilege*.
- Proposal: apply the correct / or more restrictive permissions.
  - In the above bug, only the user running the proxy server should be permitted to have read access to the configuration file

Incorrect handling of file permission errors

- CVE-2004-0148: wu-ftpd 2.6.2 and earlier, with the restricted-gid option enabled, allows local users to bypass access restrictions by changing the permissions to prevent access to their home directory, which causes wu-ftpd to use the root directory instead.
- Proposal: Introduce code that handles all errors coming from insufficient privileges (e.g. a failed call to `open(2)`) in a way that adheres to the security requirements of the project (e.g. the action will not be performed if the user lacks the required privileges).

Permission race condition during copy

- The product, while copying or cloning a resource, does not set the resource's permissions or access control until the copy is complete, leaving the resource exposed to other spheres while the copy is taking place.
- Note: data is written to a directory accessible by other spheres.
- CVE-2002-0760: Archive extractor decompresses files with world-readable permissions, then later sets permissions to what the archive specified.
- Proposal: Limit the default permissions (`umask(2)`) assigned to newly created files. Enforce the desired permissions during the creation of the file (see `mode` argument of `open(2)` in C).

Path traversal

- The software uses an improper mechanism to limit access to a specific file or set of files. An attacker can influence the path from which files are opened and can thus read or write to arbitrary locations on the filesystem.
- CVE-2009-1760: libtorrent would honor relative paths (e.g. `../.bashrc`) found in .torrent files thus allowing attackers to write/overwrite files at arbitrary locations on the user's filesystem.
- Proposal:
  1. compose the path from the trusted base (e.g. `/path/basedir`) and the untrusted input (e.g. `../../foo`). Be sure that the composed path does not exceed PATH_MAX.
  2. Apply a function such as `realpath(3)` to determine the absolute path of the file.
  3. Check whether the resulting directory and filename are considered valid for the intended operation.

Improper handling of special files

- The user is allowed to specify a non-regular file resulting into unintended program behaviour
  - Windows devices: AUX, CON, PRN, COM1, LPT1
  - Unix devices: /dev/zero, /dev/random
  - Windows ::DATA alternate data stream
  - application-provided files: /dev/tcp/4.4.4.4/80 (allows connecting to port 80 of 4.4.4.4 from bash)
- Example: Denial of service caused by reading from `/dev/zero`
- Proposal: Check the type of the file before opening the file. On POSIX systems use `stat(2)` for the check. On Windows check for special file names[1] (as the type of file is deduced by the extension).

[1] see Windows File Naming rules and NTFS reserved files

Insecure permissions and temporary files

- Temporary files are usually written in world-accessible directories (e.g. `/tmp`).
- If the temporary file **has wrong permissions**, it may be accessible by other spheres.
- If the temporary file is written **inside a directory with wrong permissions** then it may be removed or replaced by other spheres.

Insecure temporary file creation

Example:

```
// dump temporary data
fd = open("/tmp/temp", O_WRONLY | O_CREAT);
write(fd, buffer, count);
close(fd);
```

- What happens if an attacker makes `'/tmp/temp'` a symbolic link pointing to `'/home/joe/.ssh/authorized_keys'` and makes user 'joe' execute the vulnerable application ?
- A race condition!
- Proposal: Use either the `O_EXCL` mode of `open(2)` or the `mkstemp(3)` function to atomically create and return a descriptor belonging to a unique temporary file.

Part III

File writing bugs

Information leak caused by uninitialized buffer written to file descriptor

Example:

```
struct person { char name[20]; unsigned char age; };

int writeperson(int fd, char *name, unsigned char age) {
      struct person p;
      p.age = age;
      strncpy(p.name, name, 20);
      return write(fd, &p, sizeof(struct person));
}
```

- A buffer written out to a file may contain uninitialized data, exposing sensitive information found in program memory (e.g. hints about ASLR mappings, stack canaries, private keys etc.).
- Proposal: Always initialize a buffer (with `memset(3)` etc.) before writing its contents to a file. This also holds true for buffers written to sockets.

Storage resource depletion

Storage depletion case: ZIP bomb

- Example: A compressed ZIP archive contains a huge amount of zero bytes that were efficiently compressed, making the ZIP file small in size. A web service accepts to process the ZIP file because of its small size. During decompression all available disk space is used leading to a denial of service condition.
- Proposal: Use a decompression algorithm that will fail to continue once a specific amount of output bytes have been written to disk.
  - Example: see `java.util.zip.inflater.inflate(.., int len)` method.

Part IV

File parsing bugs

Information leak caused by uninitialized buffer written to file descriptor

Memory (and CPU time) consumption case: Billion Laughs attack

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ELEMENT lolz (#PCDATA)>
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

- Denial of service due to exponential entity expansion
- Proposal: `FEATURE_SECURE_PROCESSING` of SAX parsers sets `entityExpansionLimit` and `elementAttributeLimit`

Buffer Overflow

Example:

```
int size;
char header[JPEG_HDR_SIZE];
read(fd, &size, 4);
read(fd, header, size);
```

- What happens if an attacker supplies a malformed file where *size* $>$ *JPEG_HDR_SIZE* ?
- Proposal
  - Check whether the ’size’ described in the file is within the bounds described by the spec
  - If it’s not, it’s a malformed file and parsing should terminate
- Check the “Memory Corruption” lecture material for more information on the subject.

Insecure Deserialization

- Applications sometimes *serialize*[2] runtime objects (i.e. store them as a series of memory-location independent bytes) in order to:
  - **store** in a data store for later retrieval
  - **share** with clients, so that the server may process the data faster when later received by the client
  - **publish** non-trivial structures to the world (e.g. Machine Learning models)
- Deserialization comes with two risks
  - Missing Object Value Sanity Check
  - Serialization format allows for Type Descriptors

[2] Object serialization is also known as object marshalling.

Insecure Deserialization: Missing Object Value Sanity Check

- Let's imagine that a Python web application tracks user information through User class objects.

```
class User:
    is_admin = False
    username = "unknown"

    def set_username(self, username):
        self.username = username

    def set_admin_status(self, is_admin=False):
        self.is_admin = is_admin
```

Insecure Deserialization: Missing Object Value Sanity Check

- The application shares with clients the serialized form of their User object, using the Pickle serialization module.

```
u = User()
u.set_username('baxter')
u.set_admin_status(is_admin=False)
serialized = pickle.dumps(b)
```

serialized becomes


```
Hex Representation                                  |Printable Bytes |
----------------------------------------------------------------------
80 04 95 39 00 00 00 00  00 00 00 8c 04 75 73 65   |...9........use|
72 94 8c 04 55 73 65 72  94 93 94 29 81 94 7d 94   |r...User...)..}.|
28 8c 08 75 73 65 72 6e  61 6d 65 94 8c 06 62 61   |(..username...ba|
78 74 65 72 94 8c 08 69  73 5f 61 64 6d 69 6e 94   |xter...is_admin.|
89 75 62 2e                                        |.ub.|
```

Insecure Deserialization: Missing Object Value Sanity Check

- But an adversary is free to forge on the client side the username and administrative level information found in the payload.

```
80 04 95 39 00 00 00 00  00 00 00 8c 04 75 73 65  |...9.........use|
72 94 8c 04 55 73 65 72  94 93 94 29 81 94 7d 94  |r...User...)..}.|
28 8c 08 75 73 65 72 6e  61 6d 65 94 8c 06 62 61  |(..username...ba|
78 74 65 72 94 8c 08 69  73 5f 61 64 6d 69 6e 94  |xter...is_admin.|
89 75 62 2e                                       |.ub.|
```

is transformed to

```
80 04 95 39 00 00 00 00  00 00 00 8c 04 75 73 65  |...9.........use|
72 94 8c 04 55 73 65 72  94 93 94 29 81 94 7d 94  |r...User...)..}.|
28 8c 08 75 73 65 72 6e  61 6d 65 94 8c 06 6d 61  |(..username...ma|
73 74 65 72 94 8c 08 69  73 5f 61 64 6d 69 6e 94  |ster...is_admin.|
88[3] 75 62 2e                                    |.ub.|
```

[3] Notice how 0x89 became 0x88 to reflect a True boolean value.

Insecure Deserialization: Missing Object Value Sanity Check

- If the application blindly instantiates the object, incorrect privileges may be assigned to the session.

```
obj = pickle.loads(serialized)
print("username = %s" % obj.username) → username = master
print("admin_status = %s" % obj.is_admin) → admin_status = True
```

- Solution: Check each of the object members (just as you would do for uninitialized values) for their type and value. Any inconsistencies found should be treated as an error!
  - Example: *Our cookie says this is session XYZ, and the database says that this session belongs to user John who is a simple user, so why is the serialized data referring to another user or user of different privilege?*
- Alternate solution: in client-server scenarios, add session information to the serialized data and sign the serialized payload at the server, so that when later received (during a client request) the session information and signature can be validated.

Insecure Deserialization: Serialization format allows for Type Descriptors

- Serialization formats come in various forms (e.g. binary, XML, JSON etc.).
- If the serialization format (or the deserializer configuration) accepts Type Descriptors, then it is possible for an attacker to perform **remote code execution** on the system that unmarshals the data.
- The attacker will modify the serialized form, inserting a reference to a class that will be used for malicious purposes[4].
- Some serialization formats are so expressive that you can simply insert the full code to be executed!
- Malicious code execution may occur before the developer has a chance to inspect Object members (i.e. during object instantiation[5]).

[4] see ysoserial project for malicious payload generation for various framework gadgets.
[5] this used to be the case with the default serialization of objects in Java.

Insecure Deserialization: Serialization format allows for Type Descriptors

- Example: The web application has an Exec class in its `exec.py` Python module which the attacker will use as a *gadget*.

```
class Exec:
    command = "/bin/rm"
    parameter = "/tmp/temporary-output"

    # This is called on object destruction time
    def __del__(self):
        subprocess.run([self.command, self.parameter])
```

Insecure Deserialization: Serialization format allows for Type Descriptors

- The attacker first modifies a Type Descriptor in the serialized data to refer to the `Exec` class of the `exec.py` module.


```
80 04 95 39 00 00 00 00  00 00 00 8c 04 75 73 65  |...9........use|
72 94 8c 04 55 73 65 72  94 93 94 29 81 94 7d 94  |r...User...)..}.|
28 8c 08 75 73 65 72 6e  61 6d 65 94 8c 06 62 61  |(..username...ba|
78 74 65 72 94 8c 08 69  73 5f 61 64 6d 69 6e 94  |xter...is_admin.|
89 75 62 2e                                       |.ub.|
```


is transformed to


```
80 04 95 39 00 00 00 00  00 00 00 8c 04 65 78 65  |...9........exe|
63 94 8c 04 45 78 65 63  94 93 94 29 81 94 7d 94  |c...Exec...)..}.|
28 8c 08 75 73 65 72 6e  61 6d 65 94 8c 06 62 61  |(..username...ba|
78 74 65 72 94 8c 08 69  73 5f 61 64 6d 69 6e 94  |xter...is_admin.|
89 75 62 2e                                       |.ub.|
```

Insecure Deserialization: Serialization format allows for Type Descriptors

- Now the attacker may (optionally) influence how Exec is used, by changing Exec object attribute values.
  - `command="echo", parameter="Remote Code Execution"`

```
80 04 95 39 00 00 00 00  00 00 00 8c 04 65 78 65  |...9........exe|
63 94 8c 04 45 78 65 63  94 93 94 29 81 94 7d 94  |c...Exec...)..}.|
28 8c 08 75 73 65 72 6e  61 6d 65 94 8c 06 62 61  |(..username...ba|
78 74 65 72 94 8c 08 69  73 5f 61 64 6d 69 6e 94  |xter...is_admin.|
89 75 62 2e                                       |.ub.|
```

is transformed and extended to

```
80 04 95 39 00 00 00 00  00 00 00 8c 04 65 78 65  |...9........exe|
63 94 8c 04 45 78 65 63  94 93 94 29 81 94 7d 94  |c...Exec...)..}.|
28 8c 07 63 6f 6d 6d 61  6e 64 94 8c 04 65 63 68  |(..command...ech|
6f 94 8c 09 70 61 72 61  6d 65 74 65 72 94 8c[6] 15  |o...parameter...|
52 65 6d 6f 74 65 20 43  6f 64 65 20 45 78 65 63  |Remote Code Exec|
75 74 69 6f 6e 75 62 2e                           |utionub.|
```

[6] Notice how the Boolean type (0x94 0x89) type was converted to a String type (0x94 0x8c).

Insecure Deserialization: Serialization format allows for Type Descriptors

- Finally the web application will execute the `Exec` destructor once the deserialized object needs to be freed.

```
obj = pickle.loads(serialized)
print("username = %s" % obj.username)
print("admin_status = %s" % obj.is_admin)
```

gives

```
Traceback (most recent call last):
  File "unmarshal.py", line 6, in <module>
    print("username = %s" % obj.username)
AttributeError: 'Exec' object has no attribute 'username'
Remote Code Execution
```

- The malicious code is executed successfully, despite the *AttributeError*.

Insecure Deserialization: Serialization format allows for Type Descriptors

Solutions

- Avoid using at all costs deserialization frameworks that allow for Type Descriptors (or configure the framework to ignore Type Descriptors when possible).
  - For ML models in particular, avoid using Python Pickle and try the ONNX format.
- Again, use session-binding and signing in client-server scenarios.

Parser Differentials: Two independent parsers parse the same data

- CVE-2013-4787: Android 1.6 Donut through 4.2 Jelly Bean did not properly check cryptographic signatures in application packages (APK), as a zip entry that appeared twice, had its file signature checked against the signature of the first entry while the zip extraction occurred based on the contents of the second entry.
  - In this way, attackers could tamper with system packages / resources, gaining root privileges on Android.

Parser Differentials: Two independent parsers parse the same data

- When **multiple parsers** (e.g. *the signature check* and the *zip extraction* parsers) parse a document they **may treat values / errors in a different manner**. This may enable an attacker to overcome a security control.
- Similar attack on the web: HTTP Request Smuggling attack (exploiting differences in the front-end and backend server HTTP parser logic).
- Solution: Apply the langsec paradigm and **generate all parser code** from the same specification (see protobuf).

XXE - XML External Entity processing

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY xxe SYSTEM
     "file:///etc/passwd" >]><foo>&xxe;</foo>
```

- An external entity is included during the processing of the file.
- Many Java XML parsers are prone to XXE due to their default settings!
- Abused to draw file data, scan ports, send Windows process credentials to malicious service via UNC path (e.g. “\\malicious-host\D$”), DoS etc.
- Proposal
  - Enable `disallow-doctype-decl` in Xerces 2 parsers
  - Alternatively, provide a custom (whitelisting) implementation for External Entity Resolution.

Part V

Fuzz Testing

Fuzz Testing (aka Fuzzing)

- An automated technique that sends extraneous values to a piece of software and monitors whether the software will handle these well.
  - Output: a set of unique payloads that produce a crash at a different point in the program code.
  - Root cause analysis of a software crash may lead to the discovery of a security vulnerability.
- Very efficient method for finding file parsing errors.
- *Coverage-guided* fuzzers optimize their generated values so that they exercise as many different program paths as possible.
- *Context-aware* fuzzers understand the structure of the file they fuzz to yield better coverage
  - e.g. in generating PNG files, they correctly recompute the file CRC.
- Example fuzzers: afl, afl++, libfuzzer, JQF, peach

Program Instrumentation and Fuzz Testing

- If we are interested in specific types of problematic behaviours (e.g. undefined behaviour, buffer overflow etc.) we can **instrument** a piece of software so that (measured) erroneous behaviour leads to a program crash.
  - Instrumentation is usually performed by applying extra code before or during compilation (static instrumentation). However there are frameworks that apply instrumentation while loading the software (dynamic instrumentation).
- Then, we can use a fuzzer to **drive** program execution to interesting paths.
- Example static instrumentation software: Google sanitizers (AddressSanitizer, MemorySanitizer etc.)
- Example dynamic instrumentation software: DynamoRIO

On the program inputs generated by Fuzzers

- The inputs generated by fuzzers may be:
  - completely random values
  - boundary values (very small, or very large)
  - based on user templates
- Optimizing a value for a certain goal may occur through the application of a genetic algorithm.
- Finding the right value to exercise a certain path may be deduced through symbolic execution.

Part VI

Conclusions

Important Concepts

- Handling special files and file permissions
- File creation race conditions
- Path Traversal
- Information Leaks
- Resource consumption (storage, CPU, memory)
- Insecure Deserialization
- Langsec and Parser Differentials
- XXE
- Fuzz Testing and Program Instrumentation

Further Reading Material

- The Art of Software Security Assessment
- File handling issues at CWE
- Annual Language Theoretic Security Workshop (LangSec)
- Fuzzing: Brute Force Vulnerability Discovery
- The Fuzzing Book (online)
- "Using program instrumentation to identify security bugs" presentation by D. Glynos