Software Architecture Security

Brandon Edwards
drraid at gmail

Arch Vulnerability

- What is an architectural vulnerability?
- An architectural vulnerability is a vulnerability which is intrinsic to the design of the technology

Arch Vulnerability

- Not just the misuse of an API or misunderstanding an operation, but a failure in the application's foundational logic
- These vulnerabilities can be much more subtle and abstract than implementation flaws

Arch Vulnerability

- What causes architectural vulnerabilities?
- In a short, vague answer:

  The failure to consider, or fix, the security ramifications of a piece of functionality offered by the system

Arch Vulnerability

- Commonly affected:
    - Cryptography
    - Authentication schemes
    - Authorization enforcement
    - Combination of above

Arch Vulnerability Causes

- 1. The failure to consider all possible states:
  - A state or scenario where security is not built in or offered (lack of security)
  - A state where the offered security is invalidated
  - A state resulting from the interoperability with external components

Arch Vulnerability Causes

- 2. A failure in the logic or design of a security mechanisms or restraint

  (such as authentication, authorization)

  - Designers misunderstood the concept behind the security technology used
  - Designers assumed users will “play nicely”, or underestimated users technical competency

Arch Vulnerabilities

- Arguably the most important
  - Difficult to fix
  - Have devastating impact
  - Often reliably exploited
  - Can aid other (implementation) attacks

Arch Vulnerabilities

- Difficult to fix; often because:
  - Deeply rooted in the application
  - Once in place, cannot be changed due to backwards compatibility requirements
  - The byproduct of a relationship between multiple components; no one claims responsibility

Arch Vulnerabilities

- Devastating Impact:
  - Being foundational flaws, these typically represent a failure in the built-in security
  - The impact often extends to yield control or access at the highest possible privilege level

Arch Vulnerabilities

- Often reliably exploited, because:
  - They are unaffected by the volatility of external influences
    - OS dependence
    - Version dependence
    - State of memory
  - Unshielded by out-of-band protection mechanisms
  - Require less technically sophisticated exploits

Arch Vulnerabilities

- Work well in symphony
  - Several small architectural problems can quickly add up to one large pwnage
  - Small architectural bugs also aid in exploitation of implementation bugs

    Examples:
    - Architectural information disclosure, such as pointer inference aids in memory corruption bugs
    - Architectural load order + file write bug

Architectural Security

- Architectural security should be addressed during initial design
- Potential attacks should be identified and resolved as early as possible
- Proper architecture leaves room for only implementation bugs

Arch Vuln Example

- DLL hijacking
- Vulnerability happens as follows:
    - User opens SMB \\share containing fileX
    - User clicks fileX,
    - Application associated with fileX is opened
    - Application begins loading file
    - Application determines it requires additional functionality to handle fileX

Example continued..

- The specified DLL is not found locally on disk in the program or System folder..
- Application proceeds to check the current working directory for the DLL
- PROBLEM! Current working directory is now the attacker's SMB share
- Application loads attacker-controlled DLL
- Game Over

Arch Fail Example

- This example is difficult to fix
  - It may involve restructuring how the program loads files, or chooses to load dynamic functionality
  - Although it may be possible that it is relatively easy to fix per instance, being a Windows behavior, it affects many applications
  - Will likely continue to appear in more and more applications

Arch Fail Example

- Devastating: code execution
- Reliable to exploit: requires no shellcode or fancy memory manipulation; affects all modern versions of Windows
- Not-highly technical: can be exploited with a Windows share

Auditing Architecture

- Truly embodies the “think like an attacker”
- Initial thoughts..
  - Consider the scope of the application
    - What was the intent?
    - What should the application **not** allow
    - How can you make it deviate?
  - Think beyond the scope of the application
    - What was never considered?

Auditing Methodology

- To find vulnerabilities in an architecture, a complete understanding is required
- Ideally access to design/architecture documentation is available
- Even more ideally, the ability to converse with the designers
- The output generated from this exercise is also priceless for implementation review

1. Resources

- Resources that are used by the application
  - System resources, memory, disk access, etc
  - Content or user data, files, database
  - Code modules loaded by the application
  - Access or credentials, auth tokens used by or granted to the program

1. Resources continued

- All of resources combined represent every piece of access to data or functionality offered by a system
- Resources are always targets of the attacker
  - They may be the ultimate prized goal
  - Or a tool to leverage to obtain other resources
- Consider how resources can be attacked

2. Input

- Examine the input into the system
  - What type of data does the program get?
  - Where does the data come from?
  - What is the purpose?
    - Which components are influenced by this?
    - How trusted is this data?
    - Who provided it?
    - Is there a difference between who is expected to supply it vs. who is capable of supplying it?

2. Input

- Something to think about:

  ANY external influence you can provide which affects the program is INPUT

Input

- Reviewing input can be one of the fastest ways to identify an architectural vulnerability
- Example: consider a web application which performs authentication and content validation on the client-side in Javascript.

3. Output

- What type of output is generated?
  - Where does it go?
  - Who is allowed to access it?
  - What is the influence it has?
  - Who/what can influence it?
  - What does the output offer an attacker?
  - How can it be leveraged?

3. Output

- Something to think about:

  ANY observation of a program response which can measured is OUTPUT

3. Output

- Reviewing output can quickly shed light on architectural failures
- Example: it may be noticed that a program sends an encrypted message bundled with the key
- Likely indicative of an architectural failure; a lack of architecture to support proper cryptography

4. User Roles

- What type of users can exist?
  - This defines 'authentication'
  - Who are they, how do they relate
  - How do they identify themselves?
  - What are the varying privilege levels?
- Places where this is unclear or undefined may indicate authentication issues, or other vulnerabilities

5. Trust Boundaries

- Given user roles, and resources, where should boundaries lie?
  - This is what defines 'authorization'
  - How much trust is each resource granted?
  - How are users trusted?
  - Are there any unclear areas of trust?
  - Are the trust boundaries enforced uniformly?

Combine

- After reviewing each of the areas, combine observations
- Where was security not considered?
- Where does the security offered no longer apply?
- How do external components relate?

Questions?