# CWE Detail – CWE-494

## Description

The product downloads source code or an executable from a remote location and executes the code without sufficiently verifying the origin and integrity of the code.

## Extended Description

An attacker can execute malicious code by compromising the host server, performing DNS spoofing, or modifying the code in transit.

## Threat-Mapped Scoring

Score: 1.8

Priority: P4 - Informational (Low)

## Observed Examples (CVEs)

**•** CVE-2019-9534: Satellite phone does not validate its firmware image.

**•** CVE-2021-22909: Chain: router's firmware update procedure uses curl with "-k" (insecure) option that disables certificate validation (CWE-295), allowing adversary-in-the-middle (AITM) compromise with a malicious firmware image (CWE-494).

**•** CVE-2008-3438: OS does not verify authenticity of its own updates.

**•** CVE-2008-3324: online poker client does not verify authenticity of its own updates.

**•** CVE-2001-1125: anti-virus product does not verify automatic updates for itself.

**•** CVE-2002-0671: VOIP phone downloads applications from web sites without verifying integrity.

## Related Attack Patterns (CAPEC)

* CAPEC-184
* CAPEC-185
* CAPEC-186
* CAPEC-187
* CAPEC-533
* CAPEC-538
* CAPEC-657
* CAPEC-662
* CAPEC-691
* CAPEC-692
* CAPEC-693
* CAPEC-695

## Attack TTPs

**•** T1195.001: Compromise Software Dependencies and Development Tools (Tactics: initial-access)

**•** T1185: Browser Session Hijacking (Tactics: collection)

**•** T1072: Software Deployment Tools (Tactics: execution, lateral-movement)

**•** T1195.002: Compromise Software Supply Chain (Tactics: initial-access)

## Modes of Introduction

**•** Architecture and Design: OMISSION: This weakness is caused by missing a security tactic during the architecture and design phase.

**•** Implementation: N/A

## Common Consequences

**•** Impact: Execute Unauthorized Code or Commands, Alter Execution Logic, Other — Notes: Executing untrusted code could compromise the control flow of the program. The untrusted code could execute attacker-controlled commands, read or modify sensitive resources, or prevent the software from functioning correctly for legitimate users.

## Potential Mitigations

**•** Implementation: Perform proper forward and reverse DNS lookups to detect DNS spoofing. (Effectiveness: N/A)

**•** Architecture and Design: Encrypt the code with a reliable encryption scheme before transmitting. This will only be a partial solution, since it will not detect DNS spoofing and it will not prevent your code from being modified on the hosting site. (Effectiveness: N/A)

**•** Architecture and Design: Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Speficially, it may be helpful to use tools or frameworks to perform integrity checking on the transmitted code. When providing the code that is to be downloaded, such as for automatic updates of the software, then use cryptographic signatures for the code and modify the download clients to verify the signatures. Ensure that the implementation does not contain CWE-295, CWE-320, CWE-347, and related weaknesses. Use code signing technologies such as Authenticode. See references [REF-454] [REF-455] [REF-456]. (Effectiveness: N/A)

**•** Architecture and Design: Run your code using the lowest privileges that are required to accomplish the necessary tasks [REF-76]. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations. (Effectiveness: N/A)

**•** Architecture and Design: Run the code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by the software. OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows the software to specify restrictions on file operations. This may not be a feasible solution, and it only limits the impact to the operating system; the rest of the application may still be subject to compromise. Be careful to avoid CWE-243 and other weaknesses related to jails. (Effectiveness: Limited)

## Applicable Platforms

**•** None (Class: Not Language-Specific, Prevalence: Undetermined)

## Demonstrative Examples

**•** This code does not ensure that the class loaded is the intended one, for example by verifying the class's checksum. An attacker may be able to modify the class file to execute malicious code.

**•** This code does not verify that the external domain accessed is the intended one. An attacker may somehow cause the external domain name to resolve to an attack server, which would provide the information for a false database. The attacker may then steal the usernames and encrypted passwords from real user login attempts, or simply allow themself to access the application without a real user account.

## Notes

**•** Research Gap: This is critical for mobile code, but it is likely to become more and more common as developers continue to adopt automated, network-based product distributions and upgrades. Software-as-a-Service (SaaS) might introduce additional subtleties. Common exploitation scenarios may include ad server compromises and bad upgrades.