# CWE Detail – CWE-770

## Description

The product allocates a reusable resource or group of resources on behalf of an actor without imposing any restrictions on the size or number of resources that can be allocated, in violation of the intended security policy for that actor.

## Extended Description

Code frequently has to work with limited resources, so programmers must be careful to ensure that resources are not consumed too quickly, or too easily. Without use of quotas, resource limits, or other protection mechanisms, it can be easy for an attacker to consume many resources by rapidly making many requests, or causing larger resources to be used than is needed. When too many resources are allocated, or if a single resource is too large, then it can prevent the code from working correctly, possibly leading to a denial of service.

## Threat-Mapped Scoring

Score: 1.9

Priority: P3 - Important (Medium)

## Observed Examples (CVEs)

**•** CVE-2022-21668: Chain: Python library does not limit the resources used to process images that specify a very large number of bands (CWE-1284), leading to excessive memory consumption (CWE-789) or an integer overflow (CWE-190).

**•** CVE-2009-4017: Language interpreter does not restrict the number of temporary files being created when handling a MIME request with a large number of parts..

**•** CVE-2009-2726: Driver does not use a maximum width when invoking sscanf style functions, causing stack consumption.

**•** CVE-2009-2540: Large integer value for a length property in an object causes a large amount of memory allocation.

**•** CVE-2009-2054: Product allows exhaustion of file descriptors when processing a large number of TCP packets.

**•** CVE-2008-5180: Communication product allows memory consumption with a large number of SIP requests, which cause many sessions to be created.

**•** CVE-2008-1700: Product allows attackers to cause a denial of service via a large number of directives, each of which opens a separate window.

**•** CVE-2005-4650: CMS does not restrict the number of searches that can occur simultaneously, leading to resource exhaustion.

**•** CVE-2020-15100: web application scanner attempts to read an excessively large file created by a user, causing process termination

**•** CVE-2020-7218: Go-based workload orchestrator does not limit resource usage with unauthenticated connections, allowing a DoS by flooding the service

## Related Attack Patterns (CAPEC)

* CAPEC-125
* CAPEC-130
* CAPEC-147
* CAPEC-197
* CAPEC-229
* CAPEC-230
* CAPEC-231
* CAPEC-469
* CAPEC-482
* CAPEC-486
* CAPEC-487
* CAPEC-488
* CAPEC-489
* CAPEC-490
* CAPEC-491
* CAPEC-493
* CAPEC-494
* CAPEC-495
* CAPEC-496
* CAPEC-528

## Attack TTPs

**•** T1498.001: Direct Network Flood (Tactics: impact)

**•** T1499.001: OS Exhaustion Flood (Tactics: impact)

**•** T1499.003: Application Exhaustion Flood (Tactics: impact)

**•** T1498.002: Reflection Amplification (Tactics: impact)

**•** T1499.002: Service Exhaustion Flood (Tactics: impact)

**•** T1499: Endpoint Denial of Service (Tactics: impact)

## 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

**•** Operation: N/A

**•** System Configuration: N/A

## Common Consequences

**•** Impact: DoS: Resource Consumption (CPU), DoS: Resource Consumption (Memory), DoS: Resource Consumption (Other) — Notes: When allocating resources without limits, an attacker could prevent other systems, applications, or processes from accessing the same type of resource.

## Potential Mitigations

**•** Requirements: Clearly specify the minimum and maximum expectations for capabilities, and dictate which behaviors are acceptable when resource allocation reaches limits. (Effectiveness: N/A)

**•** Architecture and Design: Limit the amount of resources that are accessible to unprivileged users. Set per-user limits for resources. Allow the system administrator to define these limits. Be careful to avoid CWE-410. (Effectiveness: N/A)

**•** Architecture and Design: Design throttling mechanisms into the system architecture. The best protection is to limit the amount of resources that an unauthorized user can cause to be expended. A strong authentication and access control model will help prevent such attacks from occurring in the first place, and it will help the administrator to identify who is committing the abuse. The login application should be protected against DoS attacks as much as possible. Limiting the database access, perhaps by caching result sets, can help minimize the resources expended. To further limit the potential for a DoS attack, consider tracking the rate of requests received from users and blocking requests that exceed a defined rate threshold. (Effectiveness: N/A)

**•** Implementation: Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if the input is only expected to contain colors such as "red" or "blue." Do not rely exclusively on looking for malicious or malformed inputs. This is likely to miss at least one undesirable input, especially if the code's environment changes. This can give attackers enough room to bypass the intended validation. However, denylists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. (Effectiveness: N/A)

**•** Architecture and Design: For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server. (Effectiveness: N/A)

**•** Architecture and Design: Mitigation of resource exhaustion attacks requires that the target system either: recognizes the attack and denies that user further access for a given amount of time, typically by using increasing time delays uniformly throttles all requests in order to make it more difficult to consume resources more quickly than they can again be freed. The first of these solutions is an issue in itself though, since it may allow attackers to prevent the use of the system by a particular valid user. If the attacker impersonates the valid user, they may be able to prevent the user from accessing the server in question. The second solution can be difficult to effectively institute -- and even when properly done, it does not provide a full solution. It simply requires more resources on the part of the attacker. (Effectiveness: N/A)

**•** Architecture and Design: Ensure that protocols have specific limits of scale placed on them. (Effectiveness: N/A)

**•** Architecture and Design: If the program must fail, ensure that it fails gracefully (fails closed). There may be a temptation to simply let the program fail poorly in cases such as low memory conditions, but an attacker may be able to assert control before the software has fully exited. Alternately, an uncontrolled failure could cause cascading problems with other downstream components; for example, the program could send a signal to a downstream process so the process immediately knows that a problem has occurred and has a better chance of recovery. Ensure that all failures in resource allocation place the system into a safe posture. (Effectiveness: N/A)

**•** Operation: Use resource-limiting settings provided by the operating system or environment. For example, when managing system resources in POSIX, setrlimit() can be used to set limits for certain types of resources, and getrlimit() can determine how many resources are available. However, these functions are not available on all operating systems. When the current levels get close to the maximum that is defined for the application (see CWE-770), then limit the allocation of further resources to privileged users; alternately, begin releasing resources for less-privileged users. While this mitigation may protect the system from attack, it will not necessarily stop attackers from adversely impacting other users. Ensure that the application performs the appropriate error checks and error handling in case resources become unavailable (CWE-703). (Effectiveness: N/A)

## Applicable Platforms

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

## Demonstrative Examples

**•** The program does not track how many connections have been made, and it does not limit the number of connections. Because forking is a relatively expensive operation, an attacker would be able to cause the system to run out of CPU, processes, or memory by making a large number of connections. Alternatively, an attacker could consume all available connections, preventing others from accessing the system remotely.

**•** This example creates a situation where data can be dumped to a file on the local file system without any limits on the size of the file. This could potentially exhaust file or disk resources and/or limit other clients' ability to access the service.

**•** This example creates a situation where the length of the body character array can be very large and will consume excessive memory, exhausting system resources. This can be avoided by restricting the length of the second character array with a maximum length check

**•** In this example there is no limit to the number of client connections and client threads that are created. Allowing an unlimited number of client connections and threads could potentially overwhelm the system and system resources.

**•** N/A

**•** N/A

## Notes

**•** Relationship: This entry is different from uncontrolled resource consumption (CWE-400) in that there are other weaknesses that are related to inability to control resource consumption, such as holding on to a resource too long after use, or not correctly keeping track of active resources so that they can be managed and released when they are finished (CWE-771).

**•** Theoretical: Vulnerability theory is largely about how behaviors and resources interact. "Resource exhaustion" can be regarded as either a consequence or an attack, depending on the perspective. This entry is an attempt to reflect one of the underlying weaknesses that enable these attacks (or consequences) to take place.