Apple Security Bounty Target Flags

About Target Flags

Target Flags are a new security research capability in Apple operating systems that make it easier to objectively demonstrate your findings and determine your award eligibility. Much like Capture the Flag exercises used in security competitions, the specific flag that you capture confirms the level of exploitability you achieved — such as register control, arbitrary read/write, or code execution — and also correlates to a reward amount.

Research areas that support Target Flags are indicated with a flag (⚑) in the Apple Security Bounty Categories table and relevant examples. To qualify for the maximum reward in any category, you must provide a working exploit — and that exploit must use Target Flags if they apply to the category of your report. Research demonstrated by Target Flags is also eligible for accelerated awards, which are processed immediately after the research is received and confirmed, even before a fix is available.

Target Flags are supported in the latest versions of all Apple operating systems — iOS, iPadOS, macOS, tvOS, visionOS, and watchOS. For research areas without Target Flags, reports must include a detailed exploitability analysis.

There are currently two kinds of Target Flags built into Apple operating systems:

1. The Commpage Target Flag, which provides precise addresses and values to use in your proof of concept (PoC), resulting in predictable crash logs that clearly demonstrate achieving exploit primitives in userspace and kernel.
2. The Transparency, Consent, and Control (TCC) Target Flag, which allows you to easily demonstrate write access to the per-user or system TCC database and confirm whether your PoC has fully compromised TCC.

This table shows how you can use Target Flags to demonstrate different levels of attacker control:

Target Flag location	Attacker-control level	How to demonstrate
Commpage	Register control	In user space: Crash of vulnerable process with at least one general-purpose register set to `_COMM_PAGE_ASB_TARGET_VALUE` In kernel space: Crash of vulnerable process with at least one general-purpose register set to `_COMM_PAGE_ASB_TARGET_KERN_VALUE`
	Arbitrary read/write	In user space: Vulnerable process read from or write to `_COMM_PAGE_ASB_TARGET_ADDRESS` In kernel space: Vulnerable process read from or write to `_COMM_PAGE_ASB_TARGET_KERN_ADDRESS`
	Arbitrary code execution	In user space: Vulnerable process jump to address `_COMM_PAGE_ASB_TARGET_ADDRESS` In kernel space: Vulnerable process jump to address `_COMM_PAGE_ASB_TARGET_KERN_ADDRESS`
Transparency, Consent, and Control (TCC)	Modification of user or system TCC database	Use the `tccutil flag check` command to check both user and system TCC databases for the value of `integrity_flag`; `tccutil flag check` outputs "modified" for the TCC database that includes an `integrity_flag` that has been successfully modified to any value other than 0

Commpage Target Flag

Many exploits begin by finding primitives of three fundamental levels of control:

1. Register control: Partially or fully control the value in one or more registers in the CPU.
2. Arbitrary read or write: Read and/or write at an attacker-chosen address.
3. Code execution: Execute attacker-controlled code.

The system selects random numbers at boot and stores them in the commpage — a stable section of memory that can be read at the same address from any process. To demonstrate exploitability, you can use these values in the commpage as the contents to write into a register, the address to read from or write to, or the address to which to jump, from an attacker-controlled process.

```
// Apple Security Bounty random values
#define _COMM_PAGE_ASB_TARGET_VALUE         (_COMM_PAGE_START_ADDRESS+0x320)        // uir
```

```
#define _COMM_PAGE_ASB_TARGET_ADDRESS       (_COMM_PAGE_START_ADDRESS+0x328)        // uir
#define _COMM_PAGE_ASB_TARGET_KERN_VALUE    (_COMM_PAGE_START_ADDRESS+0x330)        // uir
#define _COMM_PAGE_ASB_TARGET_KERN_ADDRESS  (_COMM_PAGE_START_ADDRESS+0x338)        // uir
```

When you demonstrate any exploit primitive with the Commpage Target Flag, your PoC should crash and produce a crash log. Based on a quick inspection of register state, the crash log exposes the level of control your PoC achieved.

Here's an example crash log from an arbitrary read from the target address `0x08ad752109466b05` as identified with the exception address and register x8 control:

```
------------------------------------
Translated Report (Full Report Below)
------------------------------------
Process:             arb_read [1229]
Path:                /Users/Shared/arb_read
Identifier:          arb_read
Version:             ???
Code Type:           ARM-64 (Native)
Role:                Unspecified
Parent Process:      zsh [740]
Coalition:           com.apple.Terminal [369]
Responsible Process: Terminal [323]
User ID:             501

Date/Time:           2025-11-18 15:03:32.8685 -0800
Launch Time:         2025-11-18 15:03:32.8458 -0800
Hardware Model:      VirtualMac2,1
OS Version:          macOS 26.1 (25B78)
Release Type:        User

Crash Reporter Key:  75B8E020-03A0-4ABB-9CF4-EBB16751310B
Incident Identifier: 6EEEA894-821B-4E1B-BC77-E4D0D5CAE7D1

Time Awake Since Boot: 2300 seconds

System Integrity Protection: enabled

Triggered by Thread: 0, Dispatch Queue: com.apple.main-thread

Exception Type:    EXC_BAD_ACCESS (SIGSEGV)
Exception Subtype: KERN_INVALID_ADDRESS at 0x08ad752109466b05 -> 0xfffff52109466b0
Exception Codes:   0x0000000000000001, 0x08ad752109466b05

Termination Reason:  Namespace SIGNAL, Code 11, Segmentation fault: 11
Terminating Process: exc handler [1229]


VM Region Info: 0xfffff52109466b05 is not in any region.  Bytes after previous re
```

```
      REGION TYPE                    START - END         [ VSIZE] PRT/MAX SHRMOD
      commpage (reserved)        1000000000-7000000000   [384.0G] ---/--- SM=NUL
--->
      UNUSED SPACE AT END

Thread 0 Crashed::  Dispatch queue: com.apple.main-thread
0   arb_read                                    0x102dd44e4 main + 132
1   dyld                                        0x18ae8dd54 start + 7184


Thread 0 crashed with ARM Thread State (64-bit):
    x0: 0x0000000000000000   x1: 0x0000000000000000   x2: 0x00000000000120a8   x3
    x4: 0x0000000102dd4552   x5: 0x000000016d02b2a0   x6: 0x000000000000000a   x7
    x8: 0x08ad752109466b05   x9: 0x00000001f7c2e1f0  x10: 0x0000000000000002  x11
   x12: 0x00000000ffffffd  x13: 0x0000000000000000  x14: 0x0000000000000000  x15
   x16: 0x000000018b213608  x17: 0x00000001f921d2b0  x18: 0x0000000000000000  x19
   x20: 0x00000001f7c20248  x21: 0x00000001f7974dc0  x22: 0xfffffffffffffff0  x23
   x24: 0x0000000000000001  x25: 0x000000016d02b4d0  x26: 0x00000001f7c23970  x27
   x28: 0x0000000000000000   fp: 0x000000016d02b2d0   lr: 0x0000000102dd44dc
    sp: 0x000000016d02b2a0   pc: 0x0000000102dd44e4 cpsr: 0x20001000
   far: 0x08ad752109466b05  esr: 0x92000004 (Data Abort) byte read Translation fau

Binary Images:
       0x102dd4000 -        0x102dd7fff arb_read (*) <d721f8b6-0058-32c3-a1a2-e213
       0x18ae85000 -        0x18af23f63 dyld (*) <b50f5a1a-be81-3068-92e1-3554f2b0
               0x0 - 0xffffffffffffffff ??? (*) <00000000-0000-0000-0000-00000000
       0x18b212000 -        0x18b24e49f libsystem_kernel.dylib (*) <9fe7c84d-0c2b-

External Modification Summary:
  Calls made by other processes targeting this process:
    task_for_pid: 0
    thread_create: 0
    thread_set_state: 0
  Calls made by this process:
    task_for_pid: 0
    thread_create: 0
    thread_set_state: 0
  Calls made by all processes on this machine:
    task_for_pid: 0
    thread_create: 0
    thread_set_state: 0

VM Region Summary:
ReadOnly portion of Libraries: Total=597.3M resident=0K(0%) swapped_out_or_unalloc
Writable regions: Total=85.0M written=225K(0%) resident=225K(0%) swapped_out=0K(0%

                                VIRTUAL   REGION
REGION TYPE                        SIZE    COUNT (non-coalesced)
===========                     =======  =======
Kernel Alloc Once                  32K        1
MALLOC                           76.8M       10
MALLOC guard page                 3840K        4
STACK GUARD                      56.0M        1
```

```
Stack                              8176K        1
__AUTH                               35K       13
__AUTH_CONST                        202K       44
__DATA                              155K       33
__DATA_CONST                        126K       43
__DATA_DIRTY                        152K       36
__LINKEDIT                        591.4M        2
__OBJC_RO                          78.3M        1
__OBJC_RW                           2567K        1
__TEXT                              6040K       46
__TPRO_CONST                        128K        2
page table in kernel                225K        1
shared memory                        32K        1
===========                       =======  =======
TOTAL                             823.6M      240


-----------
Full Report
-----------

{"app_name":"arb_read","timestamp":"2025-11-18 15:03:33.00 -0800","app_version":"
{
  "uptime" : 2300,
  "procRole" : "Unspecified",
  "version" : 2,
  "userID" : 501,
  "deployVersion" : 210,
  "modelCode" : "VirtualMac2,1",
  "coalitionID" : 369,
  "osVersion" : {
    "train" : "macOS 26.1",
    "build" : "25B78",
    "releaseType" : "User"
  },
  "captureTime" : "2025-11-18 15:03:32.8685 -0800",
  "codeSigningMonitor" : 0,
  "incident" : "6EEEA894-821B-4E1B-BC77-E4D0D5CAE7D1",
  "pid" : 1229,
  "translated" : false,
  "cpuType" : "ARM-64",
  "roots_installed" : 0,
  "bug_type" : "309",
  "procLaunch" : "2025-11-18 15:03:32.8458 -0800",
  "procStartAbsTime" : 56724058072,
  "procExitAbsTime" : 56724293273,
  "procName" : "arb_read",
  "procPath" : "\/Users\/Shared\/arb_read",
  "parentProc" : "zsh",
  "parentPid" : 740,
  "coalitionName" : "com.apple.Terminal",
  "crashReporterKey" : "75B8E020-03A0-4ABB-9CF4-EBB16751310B",
  "appleIntelligenceStatus" : {"state":"unavailable","reasons":["deviceNotCapable"
```

```
"customerFused" : 0,
"developerMode" : 1,
"responsiblePid" : 323,
"responsibleProc" : "Terminal",
"codeSigningID" : "arb_read",
"codeSigningTeamID" : "",
"codeSigningFlags" : 570556929,
"codeSigningValidationCategory" : 10,
"codeSigningTrustLevel" : 4294967295,
"codeSigningAuxiliaryInfo" : 0,
"instructionByteStream" : {"beforePC":"6QMAkSgBAPkAAACQAKgUkQoAAJQAAACQAEgVkQcA
"bootSessionUUID" : "0635D522-4B8C-4D34-84AC-4A86F130A843",
"sip" : "enabled",
"vmRegionInfo" : "0xfffff52109466b05 is not in any region.  Bytes after previous
"exception" : {"codes":"0x0000000000000001, 0x08ad752109466b05","rawCodes":[1,6
"termination" : {"flags":0,"code":11,"namespace":"SIGNAL","indicator":"Segmenta
"vmregioninfo" : "0xfffff52109466b05 is not in any region.  Bytes after previous
"extMods" : {"caller":{"thread_create":0,"thread_set_state":0,"task_for_pid":0}
"faultingThread" : 0,
"threads" : [{"triggered":true,"id":24012,"threadState":{"x":[{"value":0},{"valu
"usedImages" : [
{
  "source" : "P",
  "arch" : "arm64",
  "base" : 4343021568,
  "size" : 16384,
  "uuid" : "d721f8b6-0058-32c3-a1a2-e2138169698d",
  "path" : "*\/arb_read",
  "name" : "arb_read"
},
{
  "source" : "P",
  "arch" : "arm64e",
  "base" : 6625447936,
  "size" : 651108,
  "uuid" : "b50f5a1a-be81-3068-92e1-3554f2be478a",
  "path" : "\/usr\/lib\/dyld",
  "name" : "dyld"
},
{
  "size" : 0,
  "source" : "A",
  "base" : 0,
  "uuid" : "00000000-0000-0000-0000-000000000000"
},
{
  "source" : "P",
  "arch" : "arm64e",
  "base" : 6629171200,
  "size" : 246944,
  "uuid" : "9fe7c84d-0c2b-363f-bee5-6fd76d67a227",
  "path" : "\/usr\/lib\/system\/libsystem_kernel.dylib",
  "name" : "libsystem_kernel.dylib"
```

```
  }
],
  "sharedCache" : {
  "base" : 6624362496,
  "size" : 5609635840,
  "uuid" : "b69ff43c-dbfd-3fb1-b4fe-a8fe32ea1062"
},
  "vmSummary" : "ReadOnly portion of Libraries: Total=597.3M resident=0K(0%) swapp
  "legacyInfo" : {
  "threadTriggered" : {
    "queue" : "com.apple.main-thread"
  }
},
  "logWritingSignature" : "18583b8d43cbbd419f69e83d56446d04dafbc2b7",
  "trialInfo" : {
  "rollouts" : [
    {
      "rolloutId" : "6761d0c9df60af01adb250fb",
      "factorPackIds" : [

      ],
      "deploymentId" : 240000009
    },
    {
      "rolloutId" : "64c025b28b7f0e739e4fbe58",
      "factorPackIds" : [

      ],
      "deploymentId" : 240000044
    }
  ],
  "experiments" : [

  ]
  }
}
```


When you submit a report of kernel or user-level privilege escalation, you must include a PoC using the Commpage Target Flag and an accompanying crash log to prove the capability you're demonstrating — and qualify for the maximum reward.

Not every crash is exploitable. For example, we generally do not reward NULL-pointer dereferences or assertion failures, even if they include user-controlled values in other registers. Your report is eligible for a reward only if we can confirm that the crash you submit is plausibly exploitable in a real-world attack. To maximize your potential reward, explain how the crash demonstrates that the vulnerability might be exploited, especially if the crash looks like an unexploitable NULL-pointer dereference or assertion failure. Include this analysis in your initial report, along with the information required for a complete and actionable report as described in the Apple Security Bounty Guidelines.

Register control

To demonstrate register control in a userland context, your report needs to include a PoC that, when run, crashes the vulnerable process, with at least one general-purpose register set to `_COMM_PAGE_ASB_TARGET_VALUE`. The register state needs to appear in the crash report of the vulnerable process, so that Apple engineers can validate that you achieved control of a general-purpose register. The same applies for kernel, while using the `_COMM_PAGE_ASB_TARGET_KERN_VALUE` instead.

To qualify for the category's maximum reward, your PoC must demonstrate full 64 bits of control. If your PoC achieves fewer than 64 but more than 32 bits of control, it is eligible for a partial reward. If your PoC can achieve fewer than 64 bits at a time and is repeatable for arbitrary addresses, make sure you include this detail in your report.

Arbitrary read/write

To demonstrate arbitrary read/write capabilities, your report needs to demonstrate the ability to make the vulnerable process read from or write to `_COMM_PAGE_ASB_TARGET_ADDRESS` or `_COMM_PAGE_ASB_TARGET_KERN_ADDRESS`, based on whether you're targeting userspace or the kernel. To demonstrate a write, write the value from `_COMM_PAGE_ASB_TARGET_VALUE` to `_COMM_PAGE_ASB_TARGET_ADDRESS`.

Here is an example PoC that demonstrates the read/write capability in userspace:

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

#define COMM_PAGE64_BASE_ADDRESS      (0x0000000FFFFFC000)
#define _COMM_PAGE_ASB_TARGET_VALUE   (COMM_PAGE64_BASE_ADDRESS + 0x320)
#define _COMM_PAGE_ASB_TARGET_ADDRESS (COMM_PAGE64_BASE_ADDRESS + 0x328)

int main(int argc, const char *argv[]) {
    uint64_t asb_value = *(uint64_t *)(_COMM_PAGE_ASB_TARGET_VALUE);
    volatile uint64_t *asb_address = *(uint64_t **)(_COMM_PAGE_ASB_TARGET_ADDRESS
    printf("[*] _COMM_PAGE_ASB_TARGET_VALUE : 0x%llx\n", asb_value);
    printf("[*] _COMM_PAGE_ASB_TARGET_ADDRESS : %p\n", asb_address);
    if (SHOW_ARBITRARY_READ) {
        printf("[!] Triggering arbitrary read!\n");
        (void)*asb_address; // Here's the crash reading from asb_address.
    } else {
        printf("[!] Triggering arbitrary write!\n");
        *asb_address = asb_value; // This is the crash writing asb_value to asb_a
    }
    return 0;
}
```

Arbitrary code execution

To demonstrate arbitrary code execution, your report needs to demonstrate the ability to make the vulnerable process jump to the address `_COMM_PAGE_ASB_TARGET_ADDRESS` or `_COMM_PAGE_ASB_TARGET_KERN_ADDRESS`. The register state needs to appear in the program's crash report, so Apple engineers can validate that you achieved control of the instruction pointer (PC register on Apple silicon).

Here is an example PoC that demonstrates the arbitrary code execution capability:

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

#define COMM_PAGE64_BASE_ADDRESS      (0x0000000FFFFFC000)
#define _COMM_PAGE_ASB_TARGET_VALUE   (COMM_PAGE64_BASE_ADDRESS + 0x320)
#define _COMM_PAGE_ASB_TARGET_ADDRESS (COMM_PAGE64_BASE_ADDRESS + 0x328)

int main(int argc, const char *argv[]) {
    uint64_t asb_value = *(uint64_t *)(_COMM_PAGE_ASB_TARGET_VALUE);
    uint64_t *asb_address = *(uint64_t **)(_COMM_PAGE_ASB_TARGET_ADDRESS);
    printf("[*] _COMM_PAGE_ASB_TARGET_ADDRESS : %p\n", asb_address);
    printf("[!] Triggering PC control!\n");
    void (*pwn)(void) = (void (*)(void))asb_address;
    pwn(); // This is the crash jumping to asb_address
    return 0;
}
```

Transparency, Consent, and Control (TCC) Target Flag

Transparency, Consent, and Control (TCC) settings help protect sensitive user data and allow users to see which apps they granted permission to access specific information, as well as grant or revoke future access. These preferences are stored in per-user and system databases, which are read-and-write protected by Full Disk Access (FDA) and System Integrity Protection (SIP), respectively.

You can easily demonstrate modifying the user or system TCC database using two new verbs added to `tccutil` — `tccutil flag check` and `tccutil flag reset`. We've added a table to both user and system TCC databases called `integrity_flag`. `integrity_flag` does not grant any permissions and has no capability except to provide a concise way for you to demonstrate overwriting or forging the TCC database.

After granting Full Disk Access to Terminal.app from the Security & Privacy pane under Settings.app, you can test this yourself using SQLite:

```
% sqlite3 TCC.db
sqlite> select * from integrity_flag;
integrity_flag|0
sqlite> INSERT OR REPLACE INTO integrity_flag (key, value) VALUES ('integrity_flag
sqlite> select * from integrity_flag;
integrity_flag|1
% tccutil flag check
User: modified
System: default
```

`tccutil flag check` checks both the user and system TCC databases for the value of `integrity_flag`. After this inspection, `tccutil flag check` outputs "modified" for the TCC database that includes an `integrity_flag` that has been successfully modified to any value other than 0.

For convenience, `tccutil flag reset` returns `integrity_flag` to value `0`. `tccutil flag reset` does not alter any other state in the user and system TCC databases. To remove all user-level TCC selections, use `tccutil reset All`. This command does not modify the `integrity_flag`.

Here is a verbose example:

```
#!/bin/bash
# tcc_flag_wrapper.sh — Demonstration script for TCC Target Flag

echo "Checking SIP status..."
sip_output=$(csrutil status 2>&1)
echo "$sip_output"

if [[ ! $sip_output == *"System Integrity Protection status: enabled"* ]]; then
    echo "ERROR: SIP is not enabled."
    exit 1
fi

echo "Resetting integrity flag..."
if ! tccutil flag reset; then
    echo "ERROR: Failed to reset integrity flag."
    exit 1
fi

echo "Running your poc here..."
./tcc_poc.sh

echo "Checking integrity flag status..."
tccutil flag check
```

Here is a minimal example:

```
#!/bin/bash
# tcc_flag_wrapper.sh — Demonstration script for TCC Target Flag
echo "Checking SIP status..."
csrutil status

echo "Resetting integrity flag..."
tccutil flag reset

echo "Running your poc here..."
./tcc_flag_poc.sh

echo "Checking integrity flag status..."
```

```
tccutil flag check
```

To qualify for the stated reward in this category and for accelerated awards, your report that demonstrates modifying the TCC database must use the new `tccutil` flag verbs to confirm impact. Use the above commands to check the state of the `integrity_flag` either in your PoC or in an accompanying video demonstration.

Apple Security Bounty Categories

Learn more >

Terms and Conditions

Learn more >

Copyright © 2026 Apple Inc. All rights reserved.