When's a program check not just a program check? When it's an exploitable security vulnerability.

An ABEND0C4 is one of the most common types of program checks encountered on z/OS, which covers a range of program interrupt codes or PIC codes including 4, 16, 17, 56, 57, 58, and 59. An ABEND0E0 is a related type of program check for cross memory scenarios including PIC codes 40, 41, 42, 43, 44, and 45. An ABEND0C4 can occur for references to either instructions or data. When it occurs for a reference to an instruction is it a sign of a bad branch, which can also be indicated by ABEND0C1, or ABEND0C6 on an odd address, which are PIC codes 1 and 6 respectively. What all these program checks have in common is that they are the only fifteen PIC codes that are often indicators of a potential security vulnerability. In hex this is '01'x, '04'x, '06'x, '10'x, '11'x, '28'x, '29'x, '2A'x, '2B'x, '2C'x, '2D'x, '38'x, '39'x, '3A'x, and '3B'x.

Background

Are all program checks signs of a security vulnerability? No, definitely not. If a program check occurs during execution of an unauthorized program, it is almost never an indication of a security vulnerability. There are always exceptions to the rule, for example if an unauthorized program is performing a network communications function, the boundary between one server and another can be almost as significant as the boundary between authorized and unauthorized code, which is why generally speaking only authorized programs should be entrusted with communications capabilities between servers. Regardless, program checks in unauthorized programs are usually not security vulnerabilities.

Does that mean all program checks in authorized programs are security vulnerabilities? No, it does not mean that either, although we are getting closer to the truth now. There are some cases where program checks in authorized programs might not be exploitable. For example, if an authorized program gets an ABEND0C4 reading from control blocks without serialization but there is no way for anyone to observe the result and it is not written to a log file or report, it is still a design flaw but not a security vulnerability. For a program check to reveal a security vulnerability, the problem that it shows needs to be exploitable.

How can we tell if a program check found in authorized code is exploitable? It's usually helpful to ask the question, what if it didn't abend? If the problem is of the type where it will always just fail reliably with a program check, it is not exploitable and is not a security vulnerability. It should be fixed but is just a bug. On the other hand, if there is any chance it might not abend, then it could indeed usually be exploited. Let's consider the different types of program checks, and the security vulnerabilities they could reveal. In each case, there are slightly different things to look for to determine if the problem is exploitable.

Fetch violations

ABEND0C4 and ABEND0E0 can both be indicators of a fetch violation. The difference between a fetch and store violation can be determine by examining the translation exception address for an ABEND0C4 or the translation exception access register number for an ABEND0E0 combined with evaluating the operands used in the instruction to determine which operand failed. The translation exception address when applicable contains two bits to indicate if it was a store or fetch violation. The '00000800'x bit indicates a fetch violation, while the '00000400'x bit indicates a store violation. If both occur at once the system usually chooses one of the two failures, somewhat at random. See the z/Architecture Principles of Operation for more details. For a description of the SDWATRAN and SDWATEAR fields see the MVS Data Areas or this link: https://www.ibm.com/docs/en/zos/3.1.0?topic=us-important-fields-in-sdwa

How can a fetch violation be a security vulnerability? The first thing to check is what storage key was used for the fetch access. For an ABEND0C4 it is usually not a security vulnerability if user key was used. For an ABEND0E0 the key might not matter if the address space or data space referenced is the problem. Regardless, if the data being fetched requires a level of authorization to access it and we program check trying to access it, the next question to ask is what was the task structure at the time? If the task was unauthorized with JSCBAUTH off, or a user TCB key in TCBPKF, then the authority to reference the system key storage or restricted address space was likely due to running under a PC or SVC. This is a problem because an unauthorized user can interrupt their task with a stimer exit, in the middle of a PC or SVC, allowing them to view the results of the faulty instruction in the RB status information. However, for authorized tasks or SRBs that is not a problem, they can't be interrupted, so it's likely not exploitable. There are other reasons a task might be uninterruptable as well, like TCB status bits or the local lock.

What kind of information could someone gain from the RB, either from a stimer exit or a parallel task? Instructions that fetch data often load it into a register or copy it into a non-fetch protected area. That means the user could see it in the register save area in the RB or determine where to look in non-fetch protected storage based on the pointer values in registers in the RB. What about a compare or test instruction like a TM or a CLC? Those instructions set a condition code based on the contents of the data being compared, which can be observed in the condition code in the RBOPSW. It might not reveal the exact value of the system key or restricted data, unless it shows a match, but it can show valuable information about it. For example, if I can find out the first character of your password is less than "b" then I know your password most likely starts with the letter "a". Bit by bit or byte by byte vulnerabilities like this can be stitched together to reveal information about a target address by a skilled malicious user.

First, we check the key, then we check the task structure, and if the data could be revealed to a user who should not be able to gain that information, we know we have a potential security vulnerability. Why do I say potential vulnerability? There are still some corner cases to consider. One case that we should talk about is program check by design, where a program accesses an address like '7FFFF000'x to purposefully cause a program check. This is uncommon, but it could be used to avoid the hassle of following standard RAS procedures with a return code, reason code, or ABEND with a component specific error explanation. Another case could involve shared memory objects, where access to the memory object can be added or removed so a program check could indicate a lack of access, and poor RAS design, but not be exploitable. There could also be a bug where a key 0 control block is read in key 5 consistently in a specific code path, which should not be exploitable because it will always fail in this scenario so it's a bug not a vulnerability.

Store violations

ABEND0C4 and ABEND0E0 could also be indicators of a store violation. The means of differentiating between a fetch violation and store violation are described above under the fetch violations section. Once you determine it is a store violation, the next thing to check is the key for an ABEND0C4 and the target address space for a ABEND0E0. If the store is to an area that unauthorized users should not be able to access, a store violation can indicate a store protection bypass, even if it occurs under an SRB or an authorized task with JSCBAUTH on or a system TCB key in TCBPKF. While authorized tasks or SRBs can not be influenced or observed directly by unauthorized users, they can overwrite storage in ways that can be taken advantage of by any unauthorized users. Store violations by authorized programs, either in system key or to a restricted address space or data space, are usually exploitable security vulnerabilities.

How could someone take advantage of a store violation by an authorized task? Imagine the program is writing '00000000'x to address '10000000'x in system key. This could be overwriting part of a critical system program, and short circuit security controls, which might allow users to compromise z/OS. Alternatively, writing to address '40404040'x in private storage could be taken advantage of even in a started task if a user is able to invoke system services provided by the started task enough times to cause address '40404040'x to be used by a particular service, or contain a particular control block, like an ACEE. While it would not be easy to take advantage of it, a store bypass by an authorized program is a problem.

The problem is worse if the store violation is not for an authorized task. The most common example is when a user calls an SVC or PC and the authorized service overwrites storage based on the user input, either at an address specified by the user or at a length or an offset influenced by the user parameters. In this scenario, the unauthorized user program, running in problem program state key eight, could still update system key storage by passing the right parameters to the SVC or PC and causing it to update it. Even if the user can not influence the address being written to it is still a severe problem, especially if they can cause it to occur. A user who observed the problem could manipulate the system to try to take advantage of it, or a lucky user could be the beneficiary of a storage overlay that causes some mayhem.

Store violations in system key are almost always a security vulnerability. Yet there are still some corner cases to consider, like failure by design, where a program writes to address zero to purposefully cause a program check. This is not common but could be used as avoid the need for following standard RAS procedures with a return code, reason code, or ABEND with a detailed component specific explanation. Another case could involve shared memory objects, where access to the memory object can be added or removed so a program check could show a lack of access, and a poor RAS design, but not be exploitable. There could also be a bug where a key 7 buffer is updated using key 1 consistently in a certain code path, which should not be exploitable because it will always fail in this scenario so it's a bug not a vulnerability.

Wild branches

An ABEND0C4 could also be an indicator of a wild branch, as could an ABEND0C1 or ABEND0C6. An ABEND0E0 can usually not indicate a wild branch because an ALET is not used for an instruction fetch. The key for an ABEND0C4 is determining if the failure was on the instruction fetch or not. This can usually be found by checking if the translation exception address matches the failing PSW. If it matches this is a clear indication the processor tried to retrieve the instruction data but could not, which means it was accessing instruction data it was not intended to access. Branching to an unintended or unexpected location is also known as a wild branch. An ABEND0C1 is another strong indicator of a wild branch. It indicates an invalid opcode was specified. Again, a clear sign of a branch to an unintended location. An ABEND0C6 can also indicate a wild branch. Although an ABEND0C6 can occur for many reasons, one of those reasons is a branch to an odd address. Valid instructions can only begin on even addresses, so a branch to an odd address means the processer is branching to an unintended or unexpected location.

What is so dangerous about branching to an unintended address? First let's check the execution state. Problem program, user key, unauthorized programs do not pose a risk to system integrity if they branch to an unintended address because they are not in an authorized state. However, if a program is running in supervisor state, system key, authorized with JSCBAUTH, or even with a PKM key mask, if it branches to an unintended or user specified program that code will also execute in the same authorized state. For more on PKM masks see: https://www.ibm.com/docs/en/zos/3.1.0?topic=routines-psw-key-mask-pkm

One way this could be taken advantage of is if an unauthorized caller of an SVC or PC can specify a parameter and the service will branch to it an authorized state. This is also known as an authorization SVC or authorization PC and is a well-known exploit for bypassing system integrity to elevate privileges. SVC or PC routines that allow this, either by design or by accident, introduce a significant security hole and should be disabled and removed from any z/OS system where they are found. Another way that a wild branch in an authorized state could be taken advantage of, even if it is not going to a caller specified address, is by observing the address it is branching to and manipulating the environment so data or even a program of interest is found at that location the next time that it occurs. For example, if a started task branches to address 'FFF000'x by mistake, a user could call system services to get data at that location, such as an enqueue name or data buffer that contains instructions they want to execute in a system key, and if that started task branches to it in system key, they just succeeded, potentially compromising z/OS.

No matter whether it is under an authorized or unauthorized task, any time a branch to an unintended address occurs in an authorized state there is a risk that unintended instructions could be executed in an authorized state. Even if a user can not influence how that occurs, if an overlay results, it might lead to other problems that a user could take advantage of such as zeroing a pointer to a control block or setting random bits that happen to be in an ACEE control block. The main reason that a branch in an authorized state might not be exploitable is if it will fail reliably every time, similarly to any store or fetch violations. If a program branches to address zero for example, address zero should always contain an opcode of zero and fail with an ABEND0C1 so there should be no way to take advantage of this. The same might be true of other addresses that contain predictable data. In some cases, a program could intentionally branch to an address containing an invalid opcode using it to indicate an error. In other cases, the storage could be inaccessible due to a bug, for example a program accidentally marked fetch protected or not executable.

Conclusion

In summary, there are a wide variety of reasons an ABEND0C1, ABEND0C4, ABEND0C6, or ABEND0E0 from an authorized program, with any type of an authorization, could indicate a security vulnerability. Understanding the reason why often boils down to understanding what the ramifications would be when it does not abend if there are circumstances when it would not abend. If it did not abend, storage that should not be referenced could be read, overwritten, or branched to in an authorized state. This could bypass fetch protection, store protection, or escalate privileges, all serious violations of system integrity. Whether or not a user can directly influence the storage address where this occurs, the question to ask is if a malicious user could still take advantage of it. Whether the difficulty level is high or low, if there is a risk it could be exploited then it is a valid security vulnerability, and the problem should be addressed.