# NY/Tampa/Dallas/Raleigh RACF User Group

## Don't be the one:

### How one line of code can compromise your system integrity

Scott Woolley
Mike Kasper
IBM z/OS Secure Engineering

May 2024

How many lines of code does it take to compromise all security and integrity on the z/OS solution stack?

One

# z/OS System Integrity

# What is System Integrity?

- Property of a system that prevents users from circumventing security mechanisms
- In z/OS, this means there is no way for an unauthorized problem program to:
  - Bypass store or fetch protection
  - Bypass RACF protection
  - Obtain control in an authorized state
- IBM will resolve any reported system integrity problem in supported releases

# What is “Authorized” on z/OS?

- Supervisor State (vs. Problem State)
- PSW Key 0-7 (vs. User Key 8-15)
  - Also known as “System Key”
- APF Authorization
  - A job step program loaded from an APF–authorized library and was link–edited with authorization code AC=1.

# Boundaries from User Programs to Authorized or Privileged Programs

- SVC and PC routines
- APF authorized programs
  - Job step programs linked AC(1)
- Program Properties Table programs
- UNIX set-user-id and set-group-id programs

# Focus on the Boundary & Specially Architected Instructions

Boundary between -

- Unauthorized Requester
- and its use of an Authorized Service (PC or SVC)

*The requester's parameters are NOT to be trusted.*

*They must be referenced in the caller's key*

Safe copy instructions -

MVCK – Move With Key
MVCSK – Move With Source Key
MVCDK – Move With Destination Key
MVCOS – Move With Optional Specifications

# Referencing User Key Storage

Why mustn't you read or write to caller-specified storage while running in system key?

It may not actually be user key storage.

- User may pass in system key storage
- Or the key of the storage could have changed
  - Time of check to time of use problem – could start as user key storage and then change to system key storage
  - Storage could be freed and replaced with system key storage

How to do safely?
- Switch to user key temporarily
- Use MVCK, MVCSK, MVCDK, MVCOS to make a safe copy

# Referencing User Key Storage

## How could this be exploited?

- Unauthorized user can cause system code to be interrupted at any time when enabled/unlocked
  - Asynchronous abends (cancel, detach, etc.)
  - Dispatcher interrupts
  - Timer interrupts
    - Exits run on same task (via IRB) and can view status of system service and alter environment of task before returning control to system service
  - TSO attention interrupts

How to do safely?
- Switch to user key temporarily
- Use MVCK, MVCSK, MVCDK, MVCOS to make a safe copy

# Vulnerability patterns on z/OS

# Vulnerability Patterns for z/OS

1. The Unintentionally Authorized PC
2. Untrusted Parms, Untrusted Regs
3. Untrusted, Indirectly Anchored Parms
4. Control Block Masquerade
5. Buffer Overflow
6. Index Out-of-Bounds

# Pattern #1: The Unintentionally Authorized PC

**Critical keyword on the ETDEF service defining a PC:**

**AKM**

**(The Authorization Key Mask)**

# Pattern #1: The Unintentionally Authorized PC

**Critical keyword on the ETDEF service defining a PC:**

**AKM**

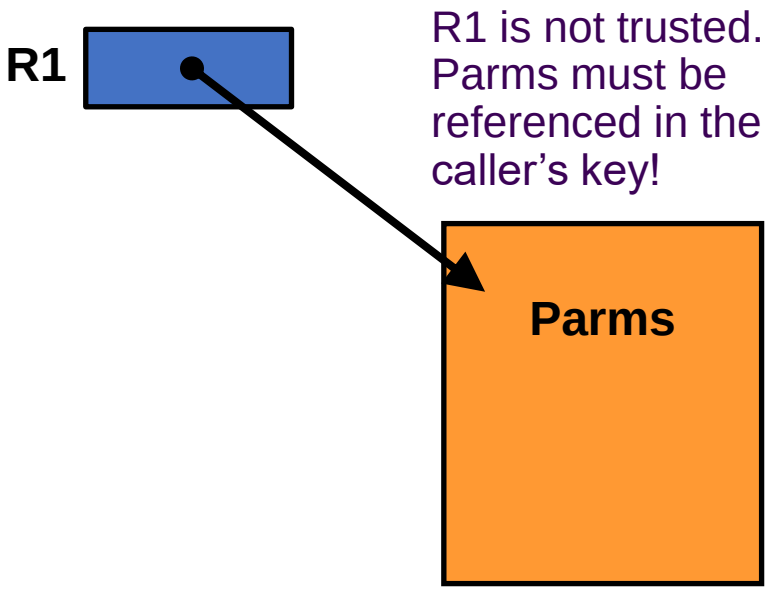
**(The Authorization Key Mask)**

**AKM(0) restricts the PC usage to callers running in key 0**

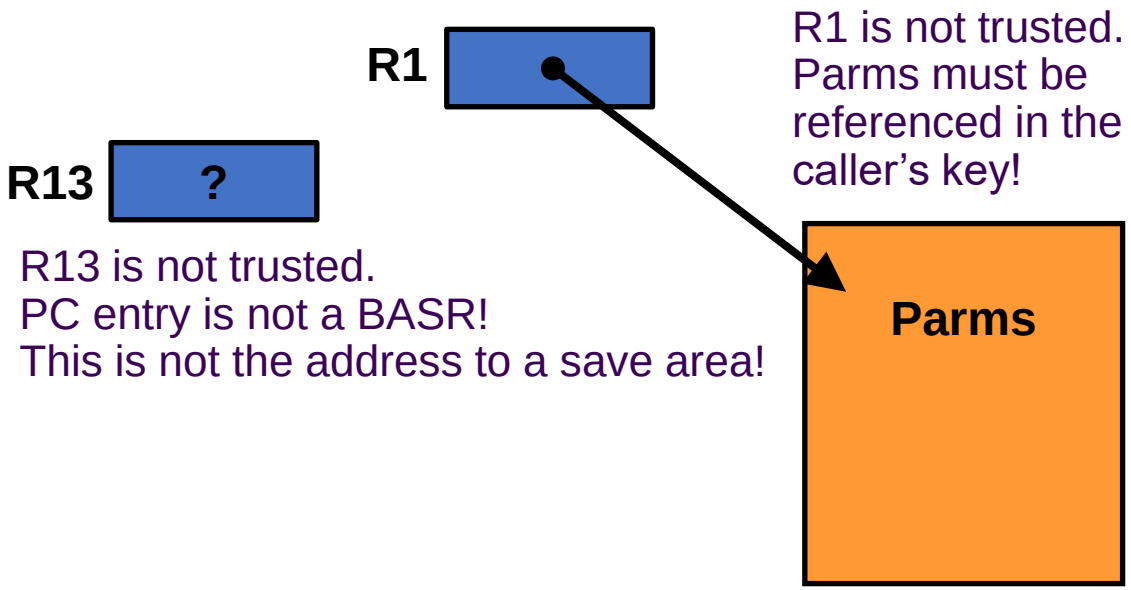
**AKM(0:15) allows the PC to be used by any caller**

If a PC target routine is *intended for authorized callers* but ***inadvertently allows unauthorized ones***, it's highly likely to have an exposure!

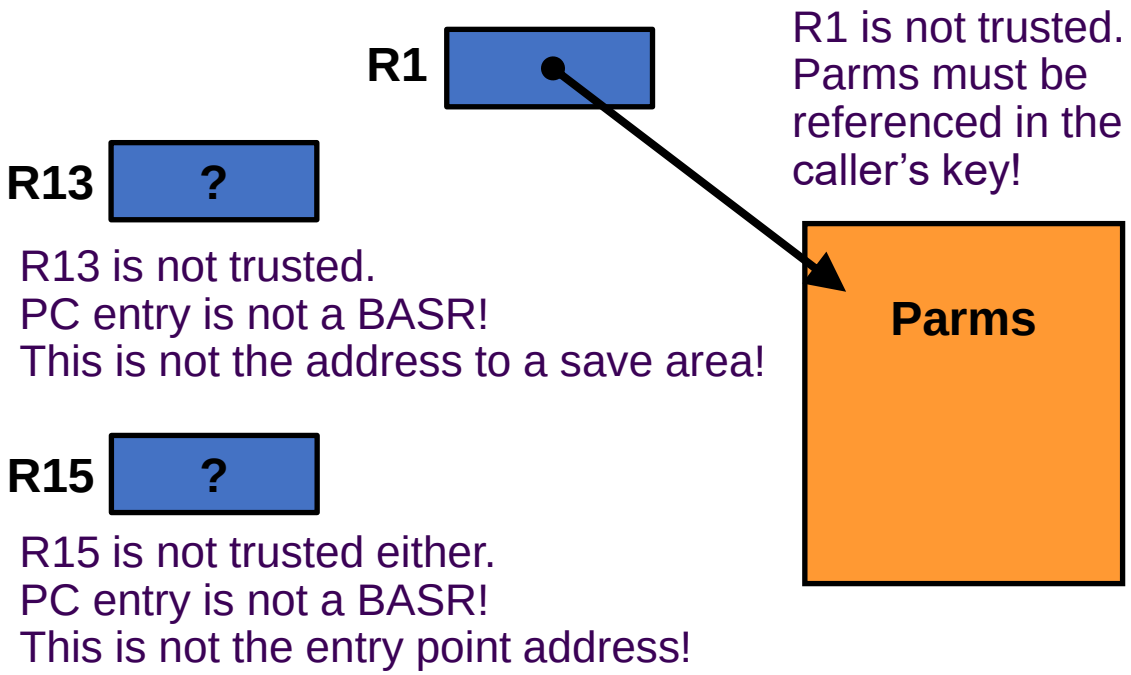
# Pattern #2: Untrusted Parms, Untrusted Regs



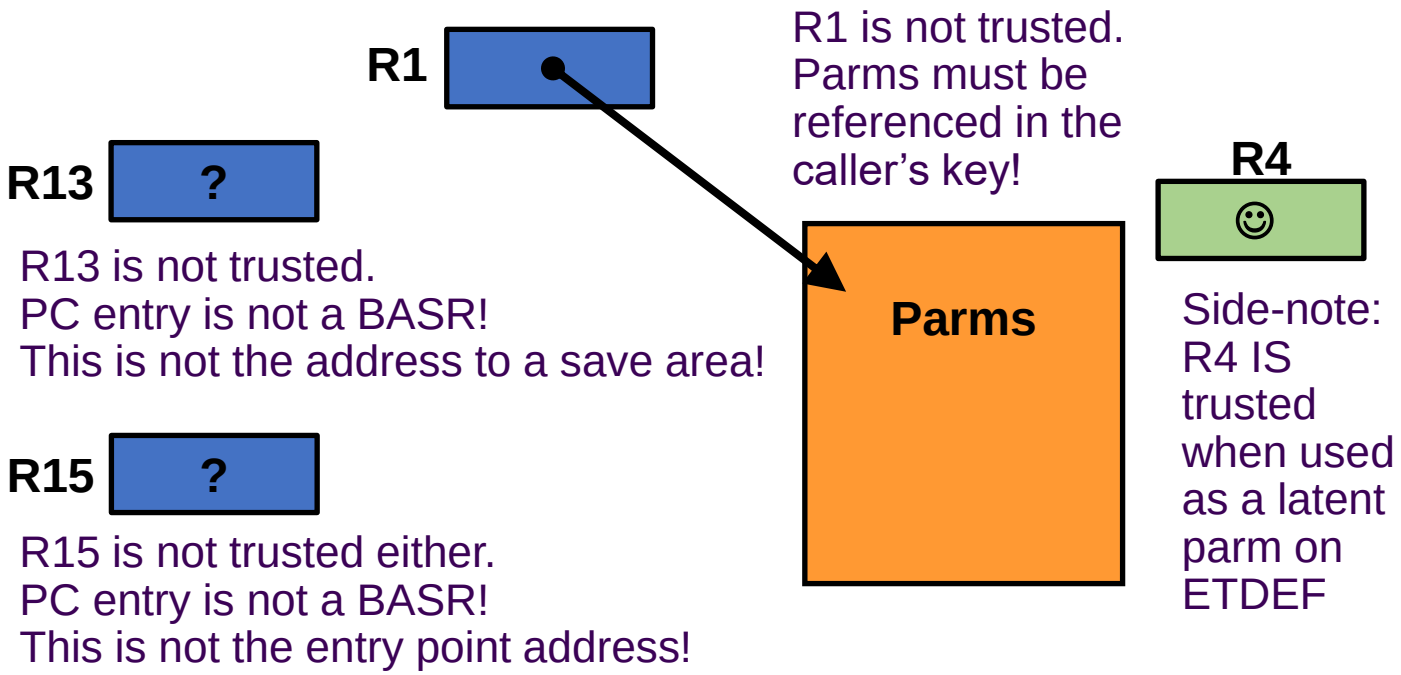
# Pattern #2: Untrusted Parms, Untrusted Regs



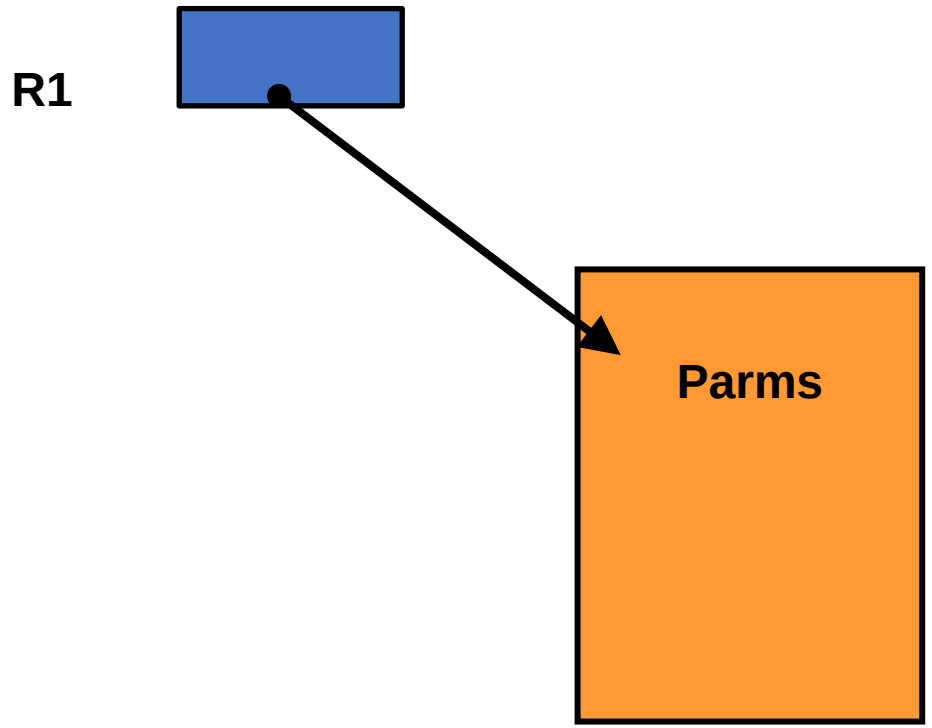
# Pattern #2: Untrusted Parms, Untrusted Regs



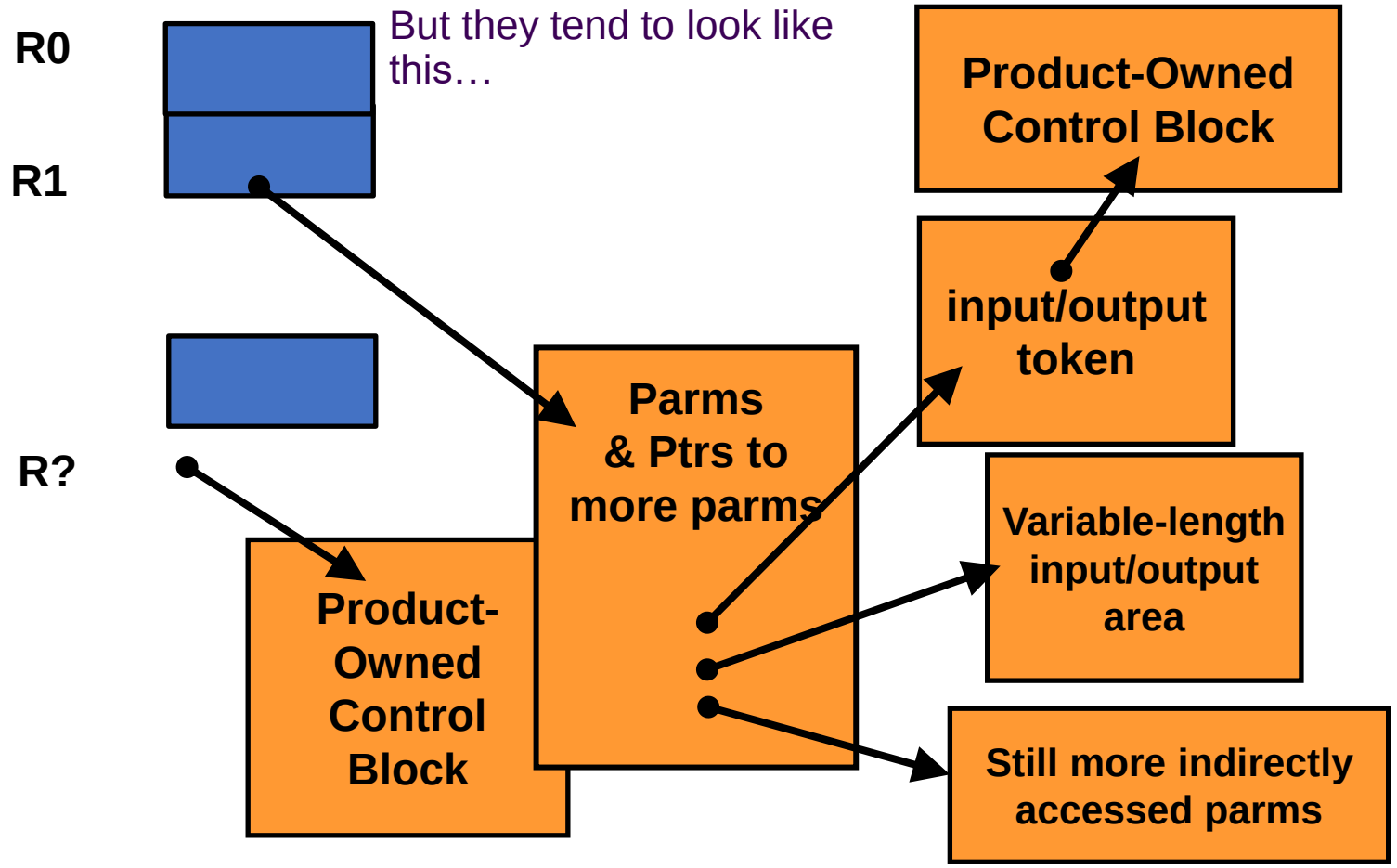
# Pattern #2: Untrusted Parms, Untrusted Regs



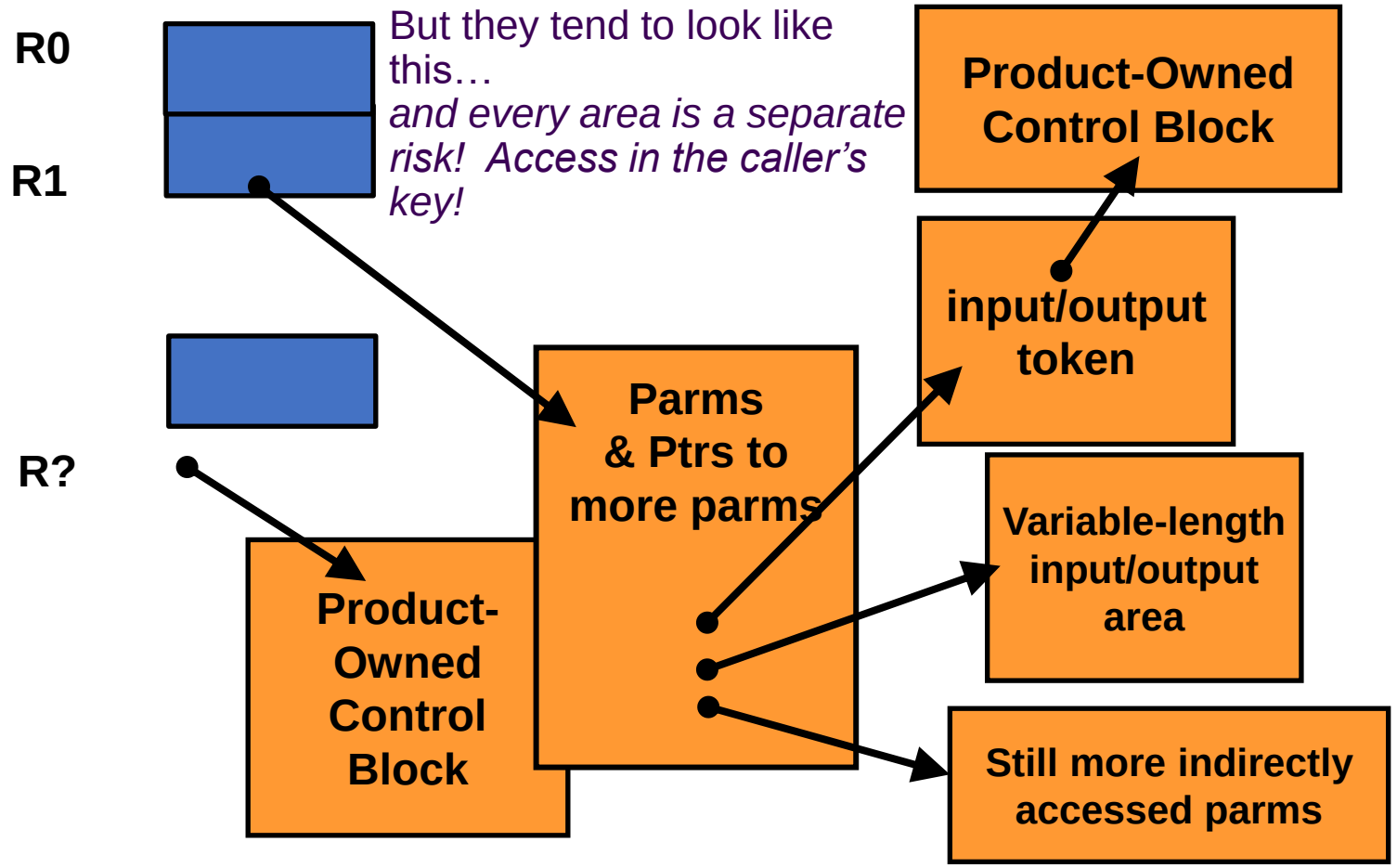
# Pattern #3: Untrusted, Indirectly Anchored Parms



# Pattern #3: Untrusted, Indirectly Anchored Parms

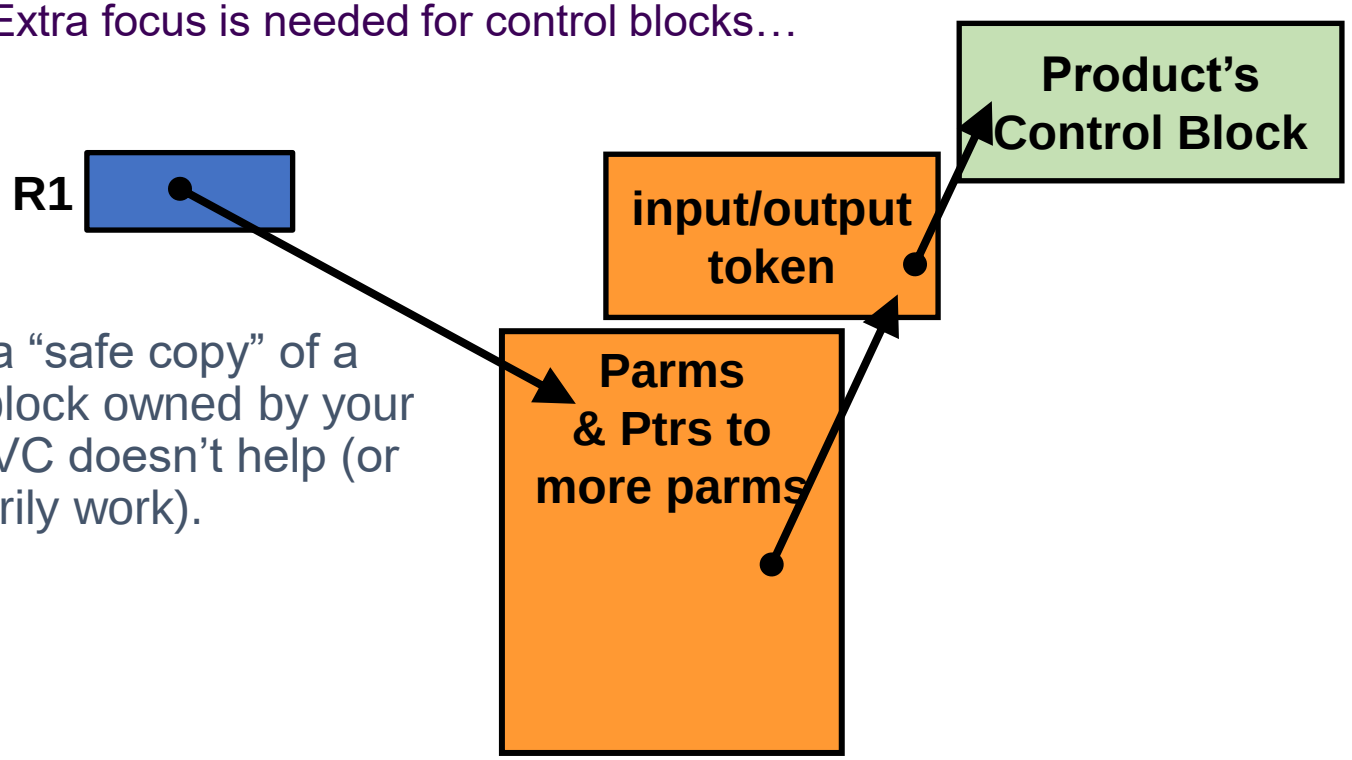


# Pattern #3: Untrusted, Indirectly Anchored Parms



# Pattern #4: Control Block Masquerade

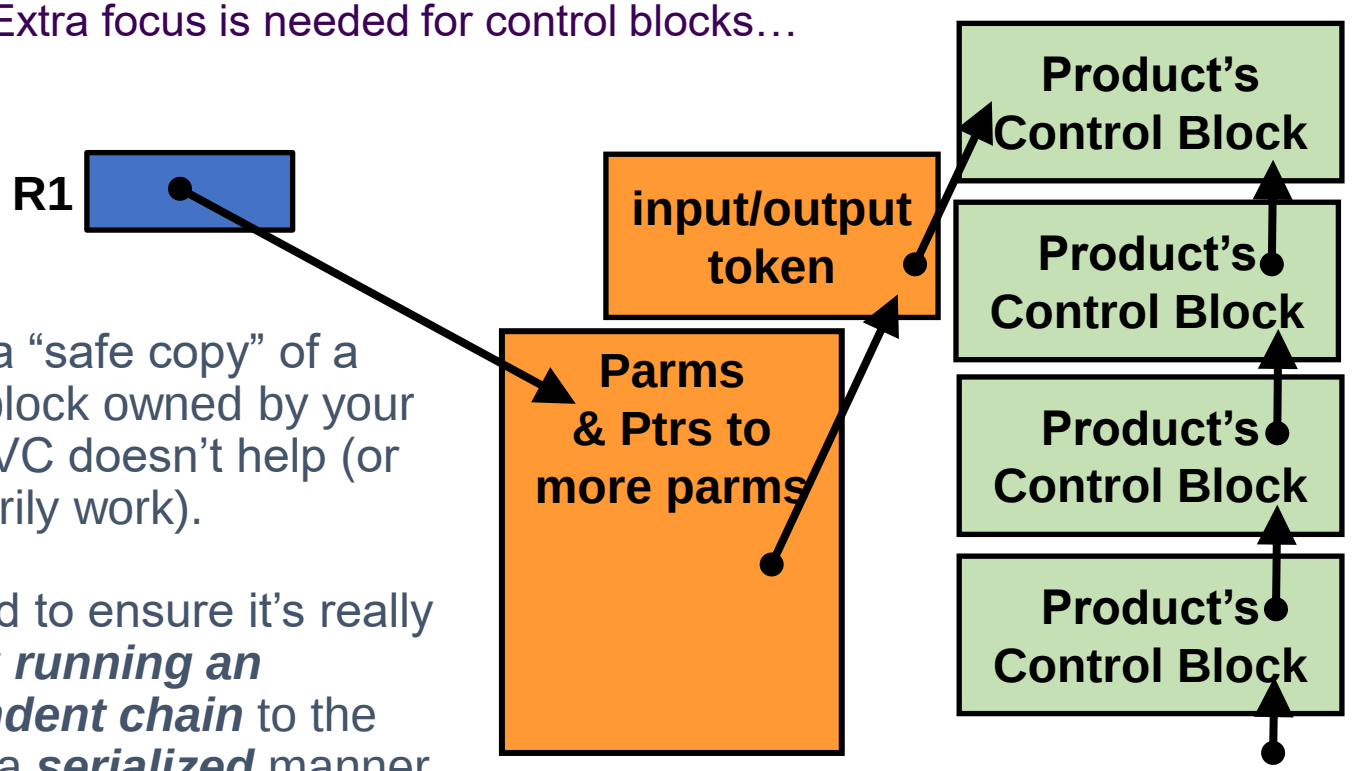
Extra focus is needed for control blocks…




Making a "safe copy" of a control block owned by your PC or SVC doesn't help (or necessarily work).

# Pattern #4: Control Block Masquerade

Extra focus is needed for control blocks…

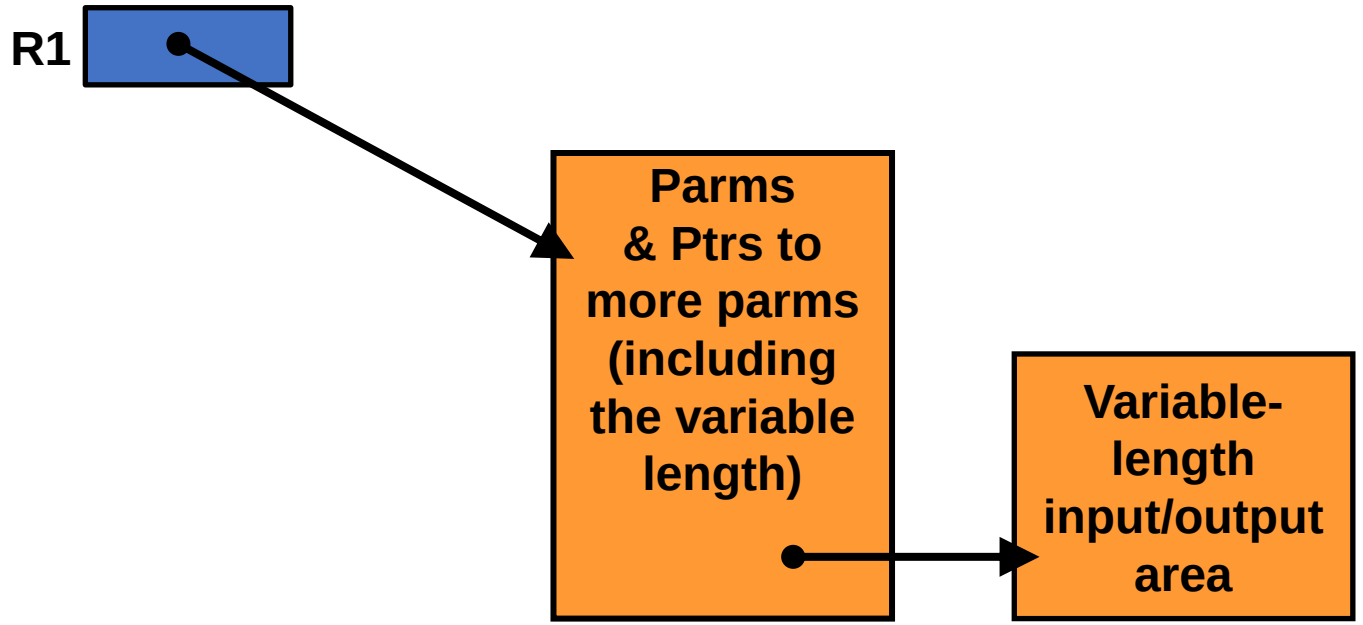



Making a "safe copy" of a control block owned by your PC or SVC doesn't help (or necessarily work).

You need to ensure it's really yours by ***running an independent chain*** to the block in a ***serialized*** manner.

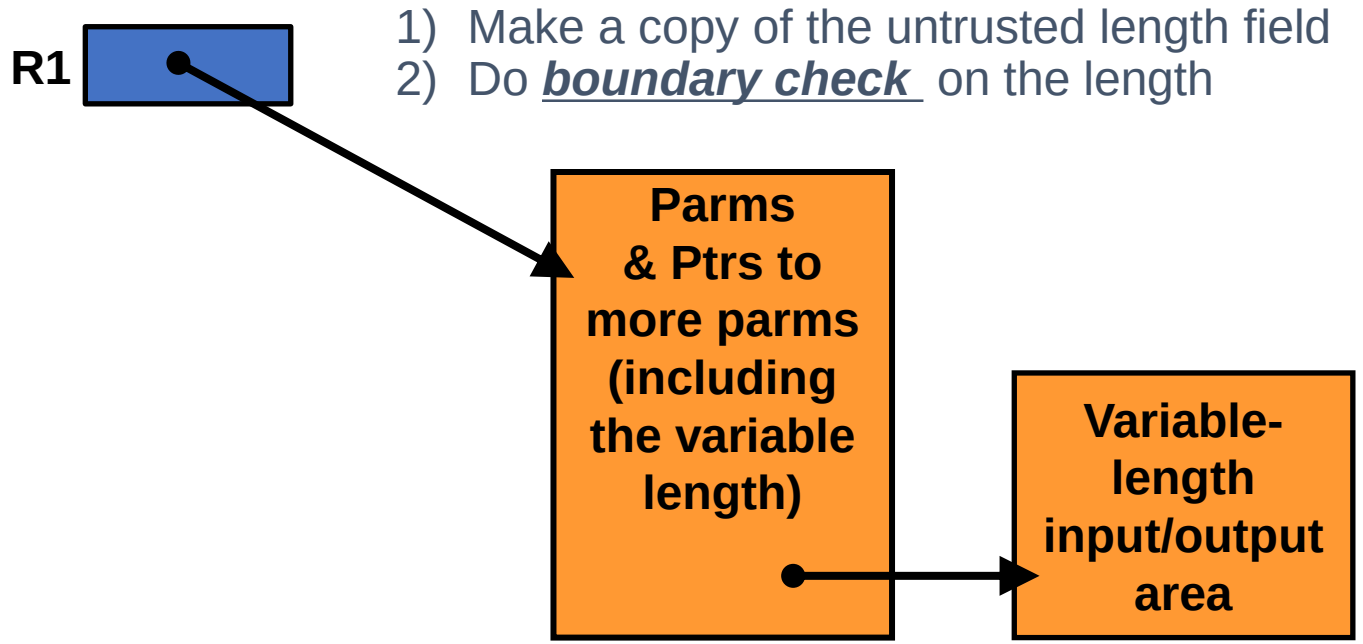
# Pattern #5: Buffer Overflow

Extra focus is also needed for variable length areas



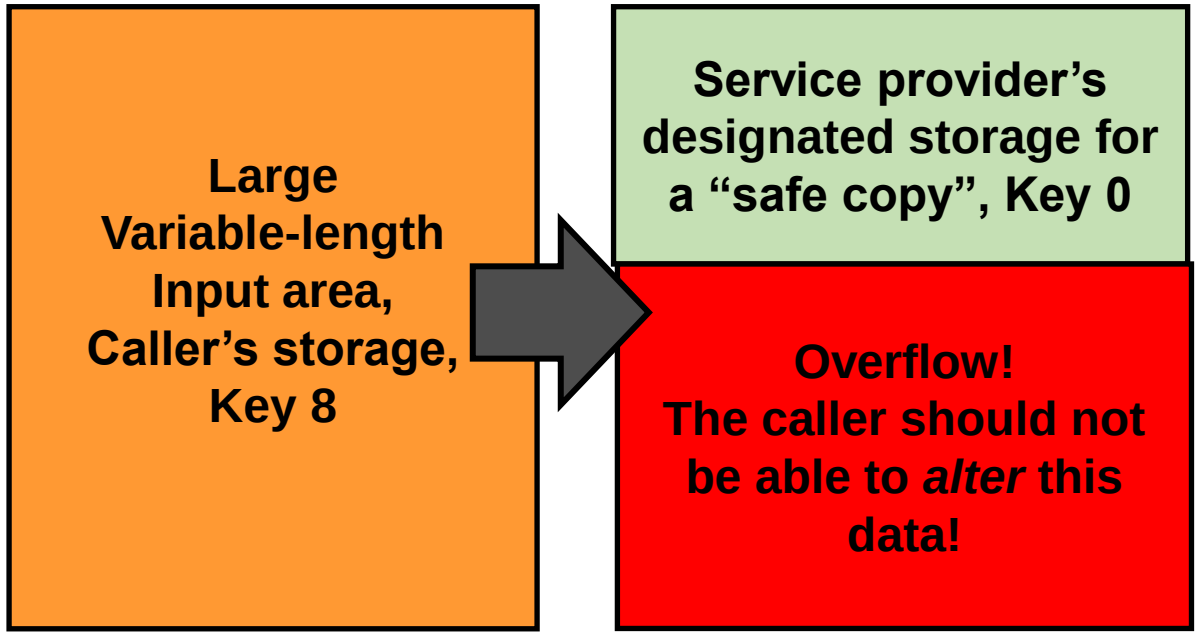
# Pattern #5: Buffer Overflow

Extra focus is also needed for variable length areas



# Pattern #5: Buffer Overflow

Clarifying the overflow of target area when copying from the caller's input area

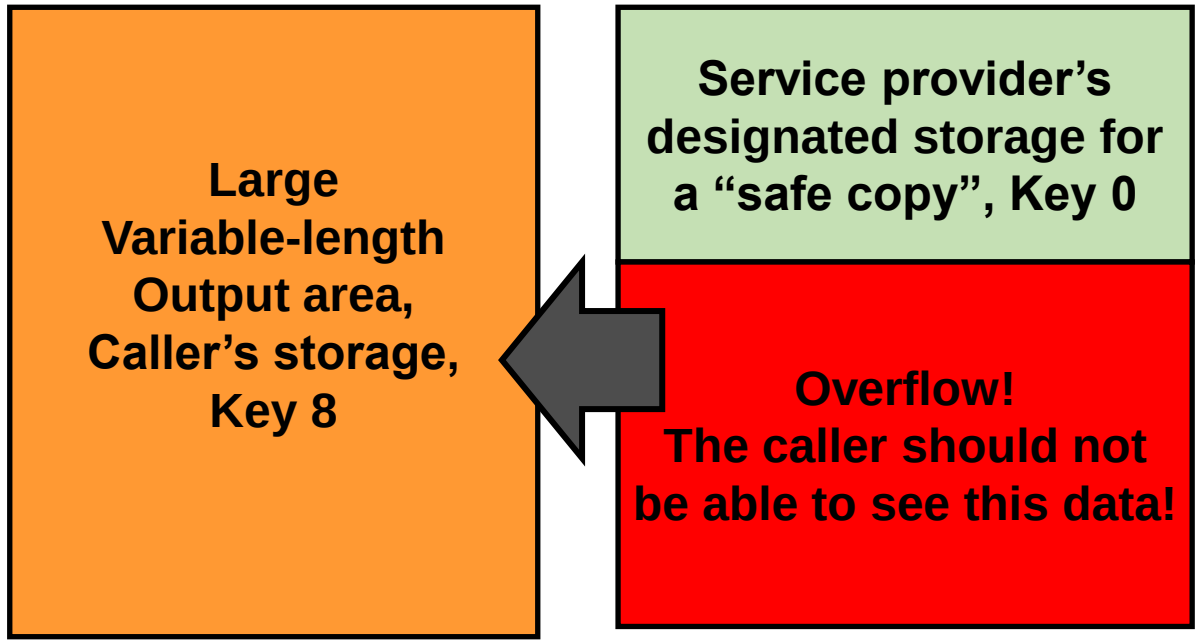



Copying with the caller's key does not protect against overflow!

Length boundary must be checked!

# Pattern #5: Buffer Overflow

Clarifying the **over-read** from source area into caller's output area

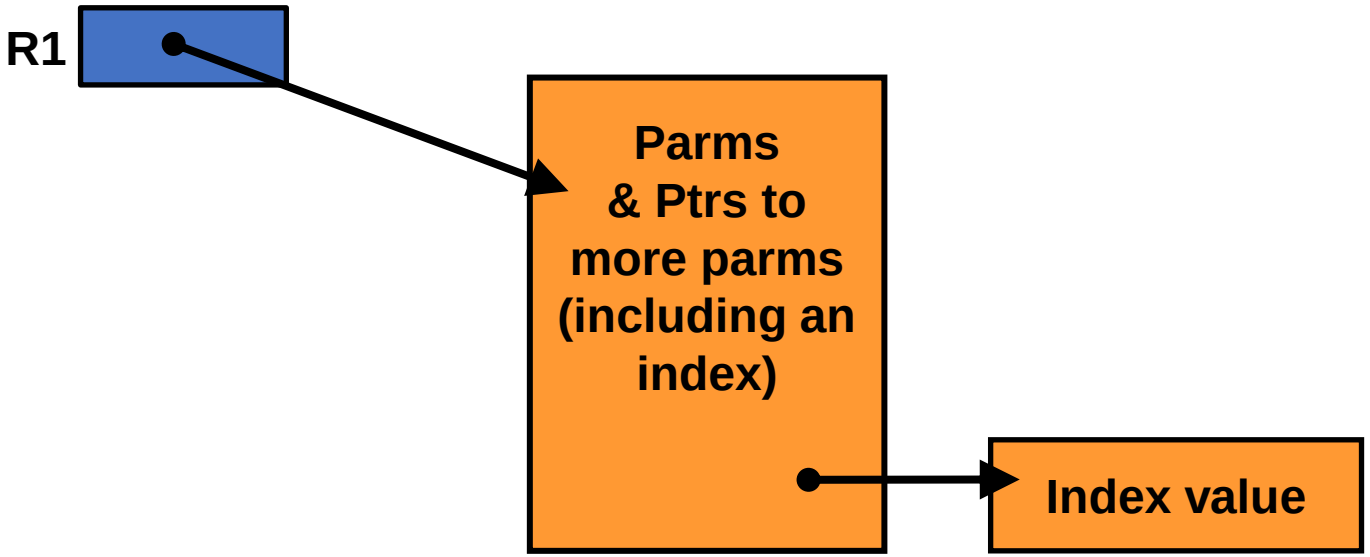



Copying with the caller's key does not protect against over-read!

Length boundary must be checked!

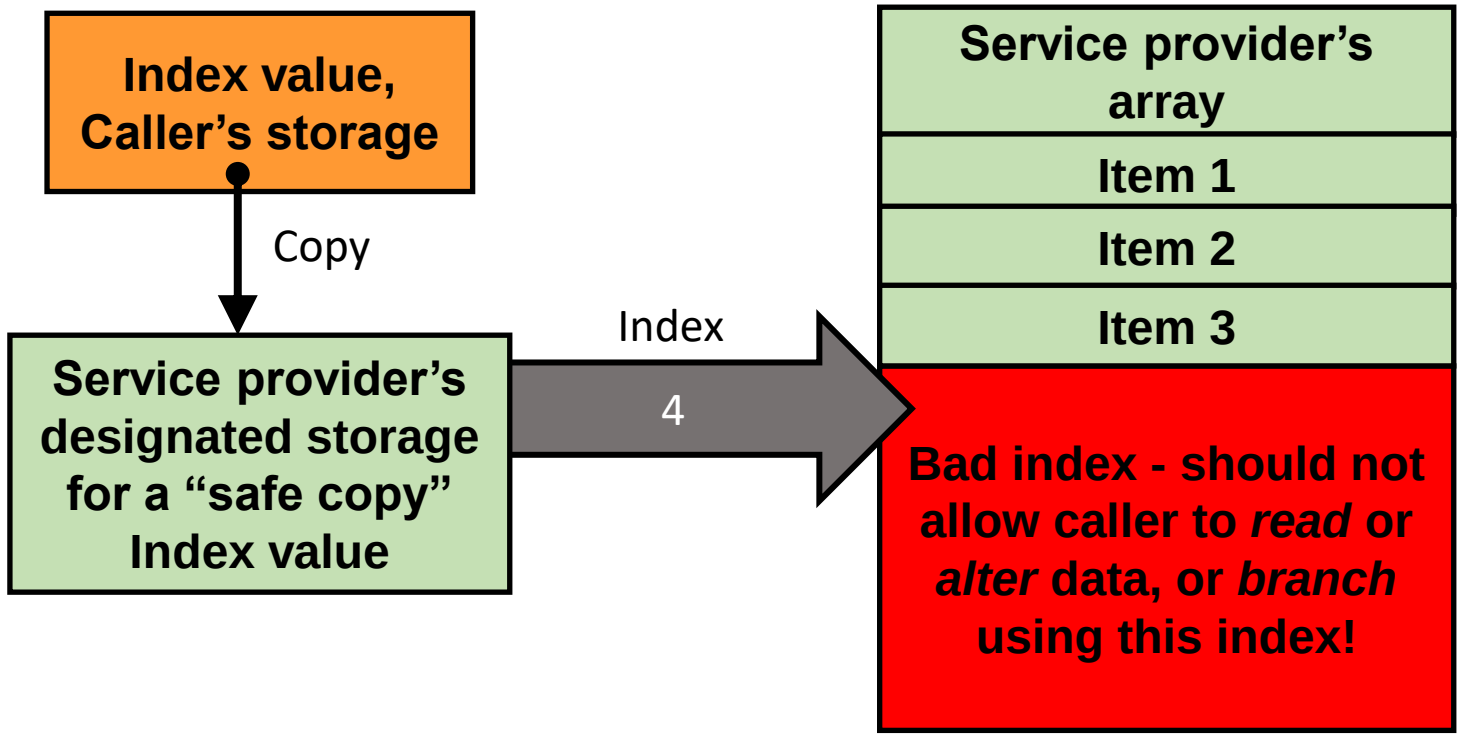
# Pattern #6: Index Out-of-Bounds

Extra focus is also needed for an index into a table or array



# Pattern #6: Index Out-of-Bounds

1) Make a copy of the untrusted index value using the key of the caller
2) Do ***boundary check*** . If index > 3 or index < 1, must reject.



# Recap: Vulnerability Patterns for z/OS

1. The Unintentionally Authorized PC
2. Untrusted Parms, Untrusted Regs
3. Untrusted, Indirectly Anchored Parms
4. Control Block Masquerade
5. Buffer Overflow
6. Index Out-of-Bounds

# What not to say about z/OS system integrity

# “I thought they checked the parameter list?”

PC routine A is available to unauthorized callers and PC routine B is only available to system key callers, but PC routine A passed a parameter from the user to PC routine B, causing PC routine B to overwrite storage in system key at the address from the user. This could still be a problem even if PC routine B is open to unauthorized callers if A runs in system key.

PC routine A was updated to obtain its own storage.

# “Why not just add the user input to the command?”

A network interface accepted a parameter with an identifier it wanted to add to a USS command, but it did not syntax check the input and used a syscall that allowed for multiple commands, so the user could append another command after the identifier which would be executed with UID 0 on the system.

Syntax checking was added and a safer syscall was used that only allows for one command to be run.

# “It’s in an APF library, so it must be safe to call.”

An authorized service allowed a user to specify any name for an exit which it would load and call, since only programs from APF libraries could be loaded, they assumed this was safe. The user specified a program that did not validate input and caused a buffer overflow giving the user’s program control.

The authorized service was updated to only use exit names if they match a system admin defined list.

# “Why would it matter if the module is reentrant?”

A PC routine open to unauthorized callers loaded a non-reentrant module and branched to it, key zero. The user set a stimer exit, overwrote the key eight code for the non-reentrant module after it loaded, and their instructions executed in PSW key zero.

The module was changed to be reentrant so that it would be loaded in key zero not user key storage.

# “We linked it as AC(1) just to be sure.”

Modules that did not expect to get control as a job step task with a parameter list were linked as AC(1), in one case because it was an alternate entry point to a load module and in another case, it was called by an AC(1) job step program.  If invoked directly, both overwrote storage using system key zero.

Both entry points were changed to no longer be AC(1) because they were not really intended to be, although it was harder for the alternate entry point.

# “My SRB doesn’t need a purge TCB or ASID.”

An SRB routine that ran in another address space was not in private storage itself but relied on control blocks and data in the home address space, so when the scheduling address space was restarted the control blocks at those locations had changed but were still being used, leading to overlays.

The SRB was updated with a purge TCB and ASID to prevent the SRB from running after those terminate.  Purge STOKEN is also available and recommended.

# “My ENF listener exit doesn’t need EOT or EOM.”

An ENF listener ran in its home address space and was loaded into private storage. It relied on control blocks and data in private storage and percolated to a recovery routine too, so when the home address space was terminated and replaced, the data and code at those locations changed but was still used.

The ENF exit was updated to add EOT and EOM yes, to stop from running after TCB or ASID termination.

# “If they used FORCE ARM, it’s their fault not ours.”

A started task was waiting on ECBs in storage that was being freed by their resource managers during address space termination. FORCE ARM led to their recovery routines getting control in unexpected ways and caused RTM to post and update ECBs in common storage that had been freed and reused.

Termination processing was updated to avoid freeing storage while it was still being used.

# “We get the storage, so no need to check the size.”

A space switching PC routine was obtaining storage in a system address space using a size specified by the user.  This allowed a PC caller to occupy all the available storage in the system address space and prevent any other requests from being processed.

The PC routine was updated to check the size first.

# “Won’t the system initialize that to zero anyways?”

An authorized program forgot to initialize some registers and storage but had lucked out and the compiler set those registers to values that led to harmless abends and the storage was never used. After a recompile the register values led to storage overlays in one case and the storage was now used in another case, so residual data was now a pointer.

The program was updated to initialize the data and registers, not rely on the compiler or residual data.

# “We can’t get serialization due to performance.”

A monitor task was running a control block chain in a target address space from an SRB it scheduled there and writing out the data out in a report for its users.  Since there was no serialization, unexpected system key data was being written out to the report.

Even if serialization might not have been practical, addition checks were added to verify the data again before anything was written to the report for users.

# “We need to display all that data for diagnostics.”

A recovery routine that received control in system key was running a save area chain based on the address in register 13 at the time of the error and displaying all the save area data for diagnostics, even if register 13 was not pointing to a save area.

The recovery routine was redesigned to stop displaying the data and just get a dump instead, unless the failing PSW was in a range that was safe.

# “No one ever complained about that program yet.”

A started task with a missing null pointer check was using data from low storage instead of the expected control block chain to find and update a control block.  If zero it would usually abend and recover, but in some scenarios the data in low storage could point to data that it would overwrite using key zero.

The started task was updated to check for zero first.

# “It recovers from the abend, so it’s not vulnerable.”

After a module was extended to add a new base register the recovery routine was not updated to restore the new base register. If an abend occurred and it retried to a point where the base register was needed it would usually abend and recover from that abend as well, but in some cases, it would overwrite unintended storage in key zero due to the unexpected value found in the base register.

Recovery was updated to restore the register.

# “Isn’t everybody using Amode31 by now?”

A new exit was added but the authorized program specifying the address of the exit forgot to turn on the high order bit, which the service it was calling used to determine which Amode to call the exit in. As a result, it cut off the high order byte and called an address below the line where a user could place their own program to get control key zero, instead.

The high order bit was turned on to fix this.

# “The POST to the missing ASID should safely ABEND”

A POST specifying an ASCB was done to an ASID that could terminate. When the ASID terminated the storage for the ECB was being freed and reused causing POST to overwrite unintended storage in the new ASID.

The Safe XM Post service IEAMSXMP was used to avoid this.

# “We need to free that code to avoid a common storage leak”

It is true that obtaining common storage every time a started task or other service is started could lead to a storage leak. However, if other address spaces could be executing that code in an authorized state, which is usually possible for modules in common storage, freeing the storage out from under them can cause unintended code to execute.

The storage was anchored to a persistent control block or in one case an authorized name token from the name token service and reused.

# Reference Material

# Links

- z/OS System Integrity Statement
  - https://www.ibm.com/downloads/cas/OWGOKG40
- IBM Z Security Portal FAQ
  - https://www.ibm.com/downloads/cas/EAO940BR
- z/OS MVS Programming: Authorized Assembler Services Guide
  - https://www.ibm.com/docs/en/zos/3.1.0?topic=guide-protecting-system

# Questions?