Secure Engineering :
How to develop authorized code without compromising z/OS system integrity

Rob Scott
Rocket Software

November 2019
Session FN




Topics Covered

WHAT IS SECURE ENGINEERING?

STORAGE PROTECTION

OBVIOUS BAD PRACTICES

PROTECTING THE UNAUTHORIZED CALLER BOUNDARY

DEFENSIVE PROGRAMMING TECHNIQUES

HELP AND REFERENCES

What is Secure Engineering?

- A methodology that protects overall system integrity
- Prevents unauthorized users from circumventing or disabling protection mechanisms
  - Obtain control in an authorized state
  - Bypass security protection
  - Bypass memory store and fetch protection
- Coding practices that prevent “system down” situations caused by poor programming techniques

Storage Protection

Storage Protection

- Each 4K frame of central storage has a key value (0-F) associated with it. This is the storage protect key.
- Storage protect keys 0-7 are reserved for z/OS and subsystem components
- Storage protect keys 8-F are reserved for normal user programs
- When a request is made to modify the contents of storage, the key associated with the request is compared to the storage protect key. If keys match, *or request key is zero*, then the request is satisfied, otherwise the request is rejected (typically 0C4-4 abend) .
- When a request is made to read the contents of the storage, the request is automatically satisfied unless the fetch protection bit is on for the frame which causes the system to use storage protection key validation as per modify.
- Typically the request key is taken from bits 8-11 of the executing PSW
  - Certain supervisor state instructions can pass the key value in a register.

Storage Protection

- Normal user programs cannot modify storage of system control blocks because of storage protection.
  - This is generally thought of as a “good thing”
- Software components (IBM and non-IBM) sometimes need to provide services for user programs that require a non-problem execution state (PSW key in the 0-7 range).
  - Typically the service is invoking other system services that are only available to authorized callers.
  - Work is carried out in an authorized state on behalf of the normal user program with the possibility of some sort of result data being relayed back.
  - This is the main battleground for system integrity and secure engineering.

Bad Practices

Obvious Bad Practices #1

- SVCs or PC routines that grant the caller elevated authority
  - A.k.a. “Magic” or “Auth On” services
  - Excuses for implementation include :
    - “But you need to pass special value only we know….”
    - “You need to know the number of the SVC/PC ….”
    - “It does some internal verification of the caller…”
- DO NOT USE.

Obvious Bad Practices #2

- Storing passwords in clear text
  - Surely this goes without saying ….
- Turning on bits in control blocks to grant privileges
  - JSCBAUTH
  - JSCBPASS
  - ACEESPEC or ACEEOPER (z/OS 2.4 now has detection mechanisms)
- Excuses include
  - “But we only do it for a short time…”
  - “We do it to make the installation easier for the customer…”
- DO NOT USE

Obvious Bad Practices #3

- Unauthorized caller loads module(s) into storage
- Module EPA stored in some unauthorized control block
- Unauthorized caller invokes authorized service that locates the control block and branches to the EPA of one of the modules
- Excuses include
  - “It was deemed too expensive to redesign the code …”
  - “Only we know the discovery logic for the control block …”
- DO NOT USE

Invalid Cross-Memory Link

- Code accesses a foreign address space STOKEN from its ASSBSTKN field
- Code issues ALESERV ADD for the ASSBSTKN using the CHKEAX=NO option of the service
- Code now expects to access storage in the foreign address space using AR mode with the ALET returned by ALESERV
- DO NOT DO THIS
    - Invalid cross-memory link
    - Code will 0C4 if source storage is paged out
    - CHKEAX basically informs z/OS to bypass normal cross-memory link check
        - *IBM have admitted that the parameter should never have been exposed to public and is used infrequently inside z/OS core by certain components in niche cases*

Protecting The Unauthorized Caller Boundary

Protecting The Unauthorized Caller Boundary

- Interfaces between the unauthorized caller and the authorized service must behave predictably.
- Includes intended and unintended interfaces.
  - It is often not what your program intended to do that caused the problem….
- Correcting dealing with
  - Caller supplied storage
  - Caller supplied values
  - Caller supplied system control blocks
- Protecting resources
  - Avoid sharing authorized data with unauthorized concurrent tasks
  - Separation and protection of caller data between other tasks
  - Serialize access to shared resources

Dealing With Caller Supplied Storage

- Only read from or write to caller supplied storage in the execution key of the caller.
  - MVCDK and MVCSK
  - MVCK
  - MODESET to caller's key
  - MVCP/MVCS
- TIP : Caller execution key can be extracted from the stack

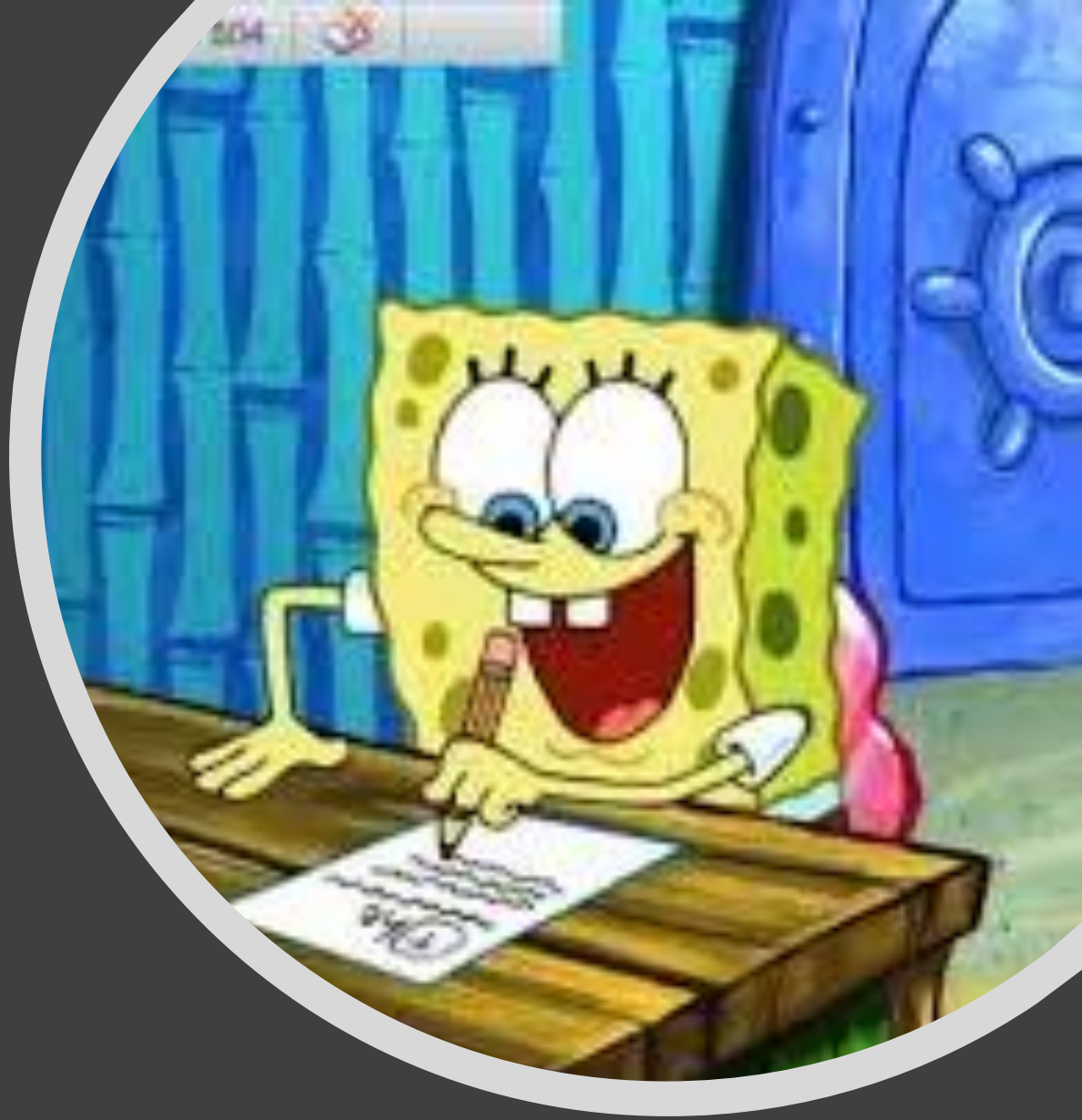
```
LHI    R1,1                    * Indicate I want PSW
ESTA   R0,R1                   * Extract from stack
N      R0,=X'00F00000'         * Just the bits I want
SRL    R0,16                   * Shift to 000000k0
```

Dealing With Caller Supplied Storage

- Always copy caller supplied parameter list and associated parameters to authorized working storage before using or validating them.
  - Ensure caller is supplying parameters in caller key
  - Ensure that values do not change after authorized service gets control
  - Copy should be performed once as close to the start of the service as possible
- Copy caller supplied parameters using key of caller : MVCSK
- TIP : Consider covering the authorised code with recovery routine to cater for the S0C4 abend that will occur when caller passes storage in incorrect key. This abend can be transposed to a more friendly return/reason code format.

Dealing With Caller Supplied Storage

- Always write to caller supplied storage in caller execution key
  - MVCDK is your friend – why not make it even friendlier and compose a macro to use it in a loop for larger amount of data?
- No exceptions
- Do NOT use SPKA/MODESET to switch into Key0 and then unconditionally write to caller supplied storage.



Dealing With User Supplied Values

- Rule #1 : Never trust user supplied values
  - Rule #2 : See Rule #1
- Verify values are legitimate
- Check for zero, null, negative, too small, too large …etc
  - You do not want your authorized code going “off the rails” because of unvalidated values – for example, any specified lengths.
- Check values for consistency if applicable
- Copy user supplied values to authorized working storage before referencing to prevent values changing in-flight

Dealing With Caller Supplied System Control Blocks

- Do NOT accept them across unauthorized/authorised caller boundary.
- Always locate system control blocks, including your own application structures, from trusted sources.
- For IBM control blocks, use the serialization techniques mentioned in the Data Area descriptions or macro prologue.
- Even for authorized/authorized caller boundary, perform validation wherever possible.
    - Many bugs are result of invalid parameter lists being passed between programs.

Examples Of Poor Design

Example Of Poor Design #1

- Client code locates product server anchor in ECSA, for example from the SSCT control block
- Client code builds parameter list requesting a function for the client userid :

```
PARMLIST            DSECT
PARMLIST_ID         DS      CL8     Eye-catcher
PARMLIST_FUNC       DS      F       Function code
PARMLIST_USERID     DS      CL8     Userid
PARMLIST_ANCHOR     DS      A       Address of anchor block
PARMLIST_ANSAREA    DS      A       Address of answer area buffer
PARMLIST_ANSLEN     DS      F       Size of answer area buffer
```

- Client locates the PC number from the product anchor and invokes it passing the address of PARMLIST via R1
- PC routine is defined by the server to execute in Key0
- PC routine accepts PARMLIST establishes addressability to the anchor block using PARMLIST_ANCHOR and then branches to a SAF security check whose EPA address is stored in the product anchor passing PARMLIST_USERID to validate the callers authority to the function code.
- THERE ARE AT LEAST THREE MAJOR SECURE ENGINEERING DESIGN FLAWS HERE.

Poor Design #1 - Flaws

- PARMLIST is referenced in Key0 and not in the caller's key
  - A safe copy of PARMLIST is never made so values could change after reference
  - Always copy to authorized working storage using something like MVCSK
- PC routine uses the anchor address passed by the client caller.
  - This could be spoofed to look like the real thing
  - Always locate system control blocks from trusted source, including the applications own anchor blocks.
- PC routine branches to module using EPA stored in the untrusted anchor
  - Caller can fool the security check
  - Caller can cause authorized PC routine to branch to callers own routine in memory, thereby elevating caller authority
- PC routine passes the userid from the caller to the security check
  - If checking callers SAF access, locate identity information from trusted IBM control blocks inside the authorized code.

Example of Poor Design #2

- Later on in the same PC routine it is time to copy some data back to the caller to the ANSAREA provided in PARMLIST.
- The address of the data to be returned is in authorized working storage field WA_ANSAREA and its size is in WA_ANSLEN.

```
LLGT  R4,PARMLIST_ANSAREA      * Get address of caller buffer
LLGT  R14,WA_ANSAREA           * Get address of data to return
LLGT  R15,WA_ANSLEN            * Load length of data
LGR   R5,R15                   * Copy length for MVCL
MVCL  R4,R14                   * Copy data back to caller
```

- THERE ARE AT LEAST TWO MAJOR SECURE ENGINEERING DESIGN FLAWS HERE.

Poor Design #2 - Flaws

- PC routine is running Key0 and the code unconditionally writes to caller storage without using caller key
- Code unconditionally writes using the length of the source data without any validation of the caller storage length.

Defensive Programming

Executing In Key0 Should Terrify You

- When running key0, you are ONE instruction away from causing an unscheduled IPL
- Storage overlay problems are one of the hardest to debug
- Quite often it is not your mainline code that causes the overlay, it will be a routine on the infrequent or recovery path
- If running in key0 then your software is under suspicion for every “system-down-because-overlay” situation.
- Every developer should suffer from “key0 anxiety”

Avoiding Key0 Anxiety

- Do not run your authorized code in key0 – use an entry in the PPT to declare your server jobstep program to another key
  - z/OS non-problem state execution key values
    - 0 – BCP
    - 1 – JES, APPC and TSO
    - 2 – unused (ISVs use this value)
    - 3 – AVM
    - 4 – unused (ISVs use this value)
    - 5 – Data management (DFP)
    - 6 – VTAM
    - 7 – DB2/MQ/IMS
- PARMLIB SCHEDxx member syntax
  - PPT PGMNAME(xyzSRV00) KEY(4) NOSWAP
  - If you are writing software for IBM, they can add your jobstep program name to the default PPT table that is shipped with z/OS (saving an install step for your customer)

Avoiding Key0 Anxiety

- Most authorized services, especially modern ones, execute in all non-problem state keys (0-7). There is no need to be specifically in key0 to call the service in most cases.
- Some developers have designed their server code to run key8 and then switch keys (via MODESET/SPKA) to key0 before invoking sections of authorized code. This is fraught with risk.
  - Storage management between key8 and key0 sections of the code is difficult.
  - Recovery/retry needs to be very aware of key switch state
- If you run key2 or key4, you do not have to keep switching keys in the server code and your storage management is much easier.
  - Running key2 or key4, you cannot overlay key0 storage
  - Your PC routines and SRBs can run in key2 or key4

Storage Subpools Matter

- Some subpools are sensitive to the key specified on STORAGE OBTAIN or GETMAIN and some are not and you get TCBKEY
  - ..and the service is completely silent about it!
- Consider the following :
  - PC-ss routine with client caller in Key8
  - PC-ss routine executes in key0 and uses the generic prologue macro for the product which gets working storage in a default subpool 50.
  - PC-ss developer tests the code and it works – “Great” they think. Job done.
    - Actually PC-ss code has, in fact, obtained and used key8 storage for its workarea because the definition of subpool 50 is that it gets TCBKEY
    - Even if the developer coded Key=0 on the prologue macro, it would be ignored.
    - Subpool 229 or 230 would have been a better choice
- Check “MVS Diagnosis Reference” for Storage Subpool properties

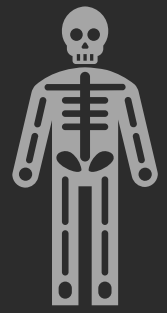
Storage Subpools Matter

Got sensitive data? Then use fetch protection!

Especially in common or shared storage

Available on STORAGE OBTAIN, GETMAIN and IARV64

Check the subpool definitions for 31-bit storage to ensure Fetch Protection is honoured. For example, subpool 229 honours fetch protection.



TIP : Do NOT use subpool 0 for anything if possible – even in unauthorized code.

Don’t be in the bucket with the world and its wife

Hard to locate storage leaks

IPCS VSMLIST is easier to use when your subpool number is uncommon.

Note – authorized callers for subpool 0 STORAGE OBTAIN get subpool 252 instead.

Good Practices

- Clear storage when system does not do it for you
  - Know the rules and/or use CHECKZERO=YES on STORAGE OBTAIN.
- Eye-catchers in control blocks
  - And verify them across interface boundaries
- Version numbers and lengths
  - Change version numbers when the length of a structure changes or there are incompatible changes to fields
  - Consider version number change when new fields introduced into reserved space
- Assert length checks
  - When control blocks should be certain sizes for design or architectural reasons
- Assert fixed offsets
  - When fields are fixed by architecture

Good Practices

- Explicitly clear memory to hex zeroes if it was used to hold sensitive data even if for only a few instructions
  - It may have been paged out
  - Someone may be running a debugger
  - Someone may see the data in an SVC DUMP
- Use EXECUTABLE=NO for STORAGE OBTAIN for data structures
  - Ensures that code cannot execute from the obtained storage
- Test on systems with DIAGxx traps and PRIMEPSA running
  - Dirty getmain and PSA contamination reveal lazy storage clear and reference
- Think beyond the API that you provide
  - Malicious usage will bypass your expected environment
  - Do you think your PC routine logic tucked away inside module ‘xyzAPI’ is hidden?

Example Exposure Due To Length Issues

- Version 1 of code obtained CSA to hold structures to enable SRB to be scheduled to another address space – overall size was 4200 bytes.
- As this size is greater than 4K, z/OS returns this CSA initialized to hex zeroes, so the developer never clears the storage.
- Storage included space for the “SRB” control block (code used the original SCHEDULE service).
- Code runs fine for many months
- Version 2 of the code included some changes that reduced the size of the CSA block to 4000 bytes.
- Code runs fine in test LPAR and is released to customers
- At a customer site on a high workload system the target address space of the SRB is fatally impacted.
- New size meant that z/OS did not initialize the storage to zeroes. On the slow test LPAR, the memory was zeroes by pure chance.
- The dirty bits the SRB control block were the cause of the fatal damage to the target address space.
- *Problem could have been avoided if some sort of assert was specified on the control block size or CHECKZERO was specified on the STORAGE OBTAIN and the storage initialized to zeroes accordingly.*

Miscellaneous Gotchas

- R13 always points to savearea … right?
  - ESTAE with NOSDWA
  - FRR
  - Always know your entry registers!
- Macro usage in incorrect environment
  - Are you sure that macro can be used in authorized code?
  - Are you sure that macro can be used in AR mode or cross-memory?
- Marking every load module AC(1) in your APF authorized load library
  - Only the jobstep program (EXEC PGM=xxxx) needs to be linked AC(1)
  - Have you tested that non-jobstep AC(1) program when put after EXEC PGM= ?
  - Maybe it will abend – but what damage might it cause due to the “surprise” of being called as jobstep?
- Copying code
  - Copying poorly designed authorized code just “because it works”

Help And References

Sample Code

- Share Fort Worth 2020
  - Session 26556 : “z/OS Cross Memory Server Code Walkthrough”
  - Sample structured assembler code for both problem state client and authorized server started task
  - How to setup a reusable System-LX and define a PC-ss routine
  - Uses IEAMSCHD to send SRB into a target address space to retrieve information on caller’s behalf
  - PLO-serialized request queue
  - 64-bit cell pool buffers in server used by PC-ss and SRB routine
  - MVCDK/SK used to transfer data between client and server
  - Code will be placed on GitHub

References

- z/OS System Integrity : Authorized Software without Security Holes
  - Karl Schmitz, IBM
- z/OS MVS Programming : Authorized Assembler Services Guide
  - SA22-7608
- MVS Planning : Security
  - GC28-1439

- My own personal battle scars …

Please submit your session feedback!

- Do it online at http://conferences.gse.org.uk/2019/feedback/FN
- This session is FN

1. What is your conference registration number?

This is the three digit number on the bottom of your delegate badge

2. Was the length of this presention correct?

1 to 4 = "Too Short" 5 = "OK" 6-9 = "Too Long"

1 2 3 4 5 6 7 8 9

3. Did this presention meet your requirements?

1 to 4 = "No" 5 = "OK" 6-9 = "Yes"

1 2 3 4 5 6 7 8 9

4. Was the session content what you expected?

1 to 4 = "No" 5 = "OK" 6-9 = "Yes"

1 2 3 4 5 6 7 8 9