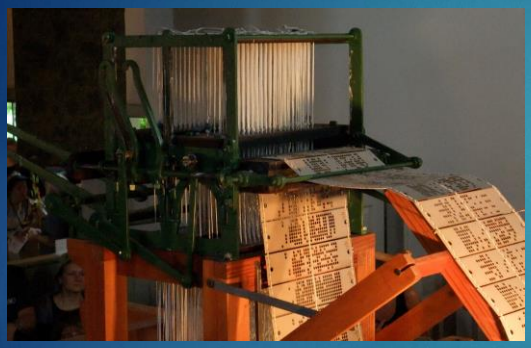
在调试器下理解ARMv8

——函数调用和栈

张银奎 2021-12-12

简史 → 离不开的行囊 → LR → 栈帧 → 溢出攻击



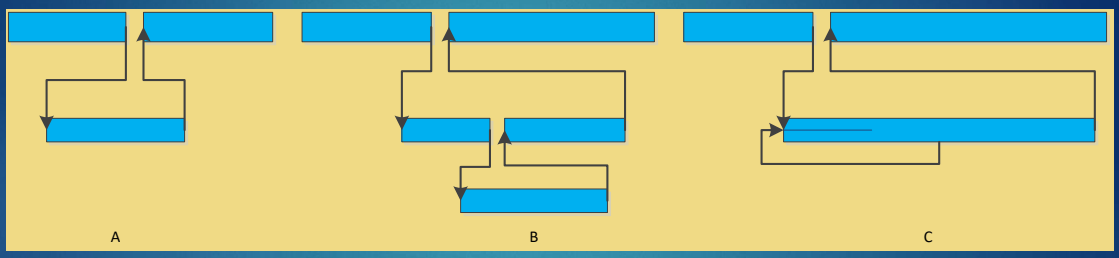
要实现分析引擎的各项目标，需要简单的方式来应付各种复杂多变的过程，像织布机那样的卡片使用模式还不够强大。我们发明了一种方法，用这种方法把卡片按照一定规则划分为若干个组备用。这样革新的目的是保证在求解一个问题时**可以把某一张或者一组卡片调出来反复使用无限多次**。

——艾达·奥古斯塔 洛甫雷斯伯爵夫人

Julian Bigelow, Herman Goldstine, J. Robert Oppenheimer, and John von Neumann在IAS(普林斯顿高等研究院）研发IAS计算机时的合影

我们把解决问题的代码序列称为程序，其中一部分被设计成可能被代入到其它程序中的形式，称为子程序。（We call the coded sequence of a problem a routine, and one which is formed with the purpose of possible substitution into other routines, a subroutine.）

——赫尔曼·戈德斯坦和约翰·冯·诺伊曼



单层调用　　多层调用　　递归调用

常记溪亭日暮，**沉醉不知归路**。
兴尽晚回舟，误入藕花深处。
争渡，争渡，惊起一滩鸥鹭。

——如梦令 李清照

Master program

Location	Order			Notes
m	A	*m*	F	The accumulator is assumed zero, so the A order adds itself into the accumulator. The A order is negative.
m+1	G	*n*	F	Jump to *n* if accumulator is negative.
m+2	. . .			The subroutine jumps here when it ends.
Subroutine				
n	A	3	F	Constructs order E (*m*+2) F from A *m* F by adding (E *m*+2 F – A *m* F) = U 2 F which is held in 3, the "spare" space of the initial orders used to hold a constant.
n+1	T	*p*	F	Stores E *m*+2 F in location *p*.
	. . .			
	. . .			
	. . .			
p	E	(*m*+2)	F	Jumps back to master program if accumulator is positive or zero.

惠勒跳转（Wheeler Jump）法

在英国剑桥大学数学实验室设计的EDSAC(Electronic Delay Storage Automatic Calculator）上发明和使用，1949开始运行

David John Wheeler
(1927 - 2004)

栈

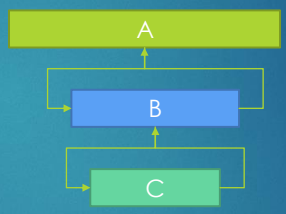
8

- 图灵发明了栈
  - 但却没有命名
  - Dijkstra命名
- 今天的大多数计算机都是基于栈架构的
  - 从CPU到OS
  - 必须依赖栈

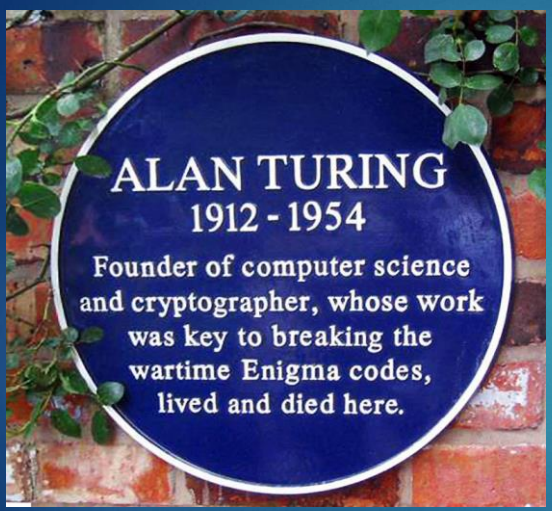
9

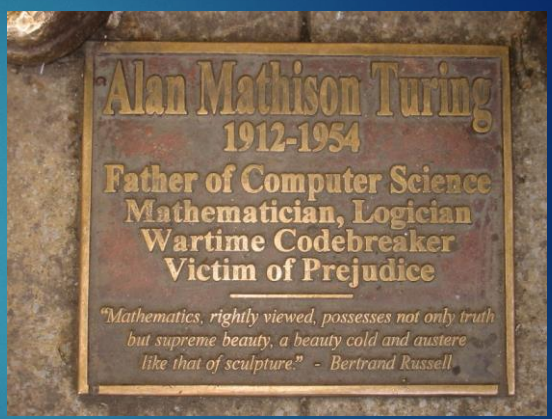
BURY
    tmpIP <- IP+DEST
    push(IP+1)
    IP <- tmpIP

UNBURY
    IP <- pop()



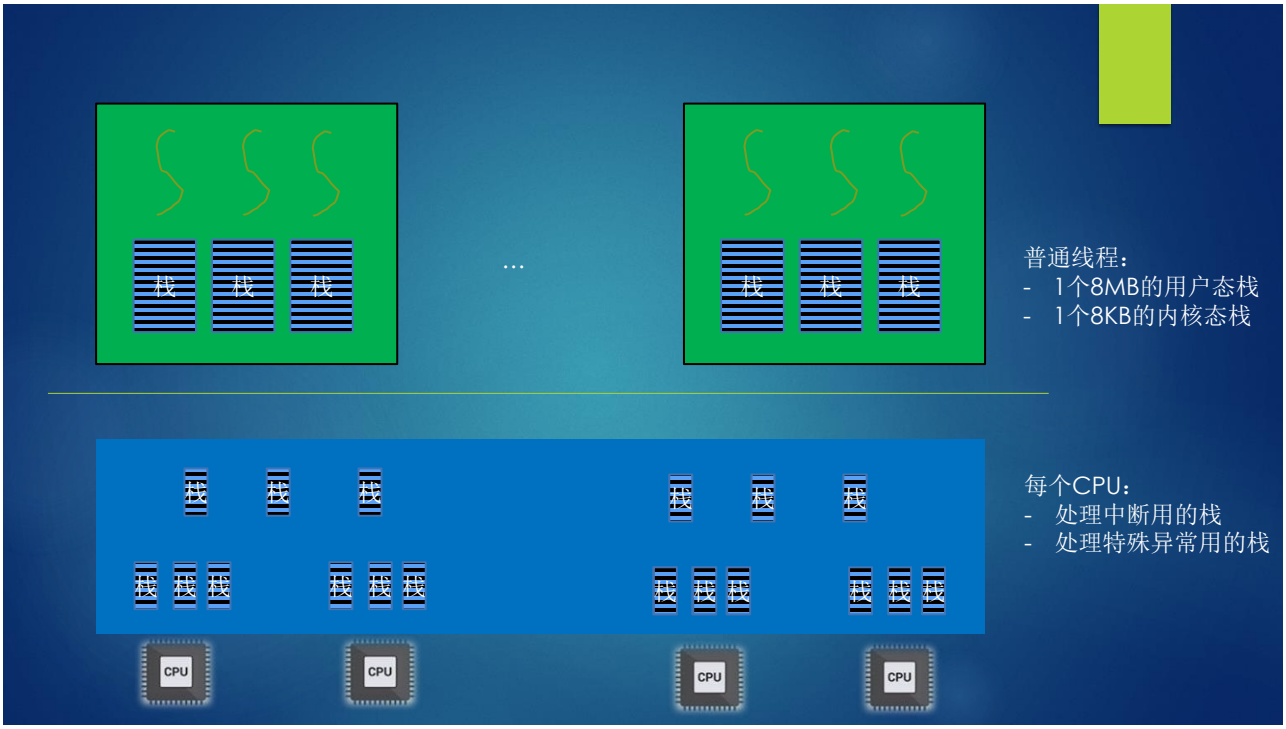
10





简史 → 离不开的行囊 → LR → 栈帧 → 溢出攻击

现代CPU离不开栈，调用子函数，传递参数等都需要栈，栈是CPU的贴身行囊



Panic现场的栈信息

案例

内核启动1.5秒时发生

```
[    1.548169] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
[    1.548473] rockchip-iodomain ff100000.syscon:io-domains: Looking up vcc1o1-supply from device t
[    1.549801] CPU: 2 PID: 1 Comm: swapper/0 Not tainted 4.4.179 #174
[    1.550378] Hardware name: Rockchip RK3328 EVB avb (DT)
[    1.550874] Call trace:
[    1.551141] [<ffffff8008088268>] dump_backtrace+0x0/0x220
[    1.551673] [<ffffff80080884ac>] show_stack+0x24/0x30
[    1.552164] [<ffffff80083b346c>] dump_stack+0x94/0xbc
[    1.552648] [<ffffff8008160c10>] panic+0xe4/0x238
[    1.553113] [<ffffff8008e211d0>] mount_block_root+0x22c/0x298
[    1.553663] [<ffffff8008e213c8>] mount_root+0x70/0x80
[    1.554146] [<ffffff8008e21578>] prepare_namespace+0x1a0/0x1b0
[    1.554701] [<ffffff8008e20dd0>] kernel_init_freeable+0x1e8/0x220
[    1.555280] [<ffffff8008a1edc0>] kernel_init+0x18/0x100
[    1.555785] [<ffffff8008082ef0>] ret_from_fork+0x10/0x20
```

miss_root_option.txt

20多秒后发生

```
[   24.148155] CPU: 2 PID: 1 Comm: swapper/0 Not tainted 4.4.179 #174
[   24.148733] Hardware name: Rockchip RK3328 EVB avb (DT)
[   24.149231] task: ffffffc00a298000 task.stack: ffffffc00a2b0000
[   24.149797] PC is at watchdog_timer_fn+0x170/0x2e0
[   24.150258] LR is at watchdog_timer_fn+0x170/0x2e0
[   24.150711] pc : [<ffffff800812bcec>] lr : [<ffffff800812bcec>] pstate: 600001c5
[   24.151420] sp : ffffffc0fef4be00
[   24.151746] x29: ffffffc0fef4be00 x28: 0000000000000003
[   24.152272] x27: ffffff8008f27720 x26: ffffff8008ef8238
[   24.152797] x25: ffffffc0fef4e250 x24: ffffffc00a2b3b30
[   24.153318] x23: 0000000000000000 x22: 0000000000000000
[   24.153837] x21: ffffff8008f26000 x20: ffffff8008ef82b0
[   24.154357] x19: ffffff8008ef8000 x18: ffffff808909f857
[   24.154876] x17: 000000000000000a x16: 0000000000000000
[   24.155395] x15: 0000000000000000 x14: 0000000000023b89
[   24.155915] x13: 000000000000000a x12: 0000000000000030
[   24.156442] x11: 00000000fffffffe x10: ffffff800909f85f
[   24.156963] x9 : 0000000005f5e0ff x8 : ffffff8008354b3c
[   24.157482] x7 : ffffff8008f8d3f8 x6 : 0000000000000035
[   24.158000] x5 : 0000000000000000 x4 : 0000000000000000
[   24.158519] x3 : 00000040f6056000 x2 : 00000040f6056000
[   24.159039] x1 : 0000000000000002 x0 : 0000000000000026
```

```
[   24.281095] Call trace:
[   24.281347] Exception stack(0xffffffc0fef4bc20 to 0xffffffc0fef4bd50)
[   24.281958] bc20: ffffff8008ef8000 0000008000000000 ffffffc0fef4be00 ffffff800812bcec
[   24.282702] bc40: 00000000600001c5 0000000000000000 ffffff80090a00b8 0000000000000000
[   24.283450] bc60: ffffffc0fef4bc80 ffffff80080ecf70 0000000000000026 0000000000000000
[   24.284196] bc80: ffffffc0fef4bd20 ffffff80080ed1d8 ffffffc0fef4bd80 ffffff8008cb10c5
[   24.284943] bca0: ffffff8008f26000 0000000000000000 0000000000000000 ffffffc00a2b3b30
[   24.285691] bcc0: ffffffc0fef4e250 ffffff8008ef8238 0000000000000026 0000000000000002
[   24.286436] bce0: 00000040f6056000 00000040f6056000 0000000000000000 0000000000000000
[   24.287185] bd00: 0000000000000035 ffffff8008f8d3f8 ffffff8008354b3c 0000000005f5e0ff
[   24.287929] bd20: ffffff800909f85f 00000000fffffffe 0000000000000030 000000000000000a
[   24.288679] bd40: 0000000000023b89 0000000000000000
[   24.289149] [<ffffff800812bcec>] watchdog_timer_fn+0x170/0x2e0
[   24.289704] [<ffffff80081021d8>] __hrtimer_run_queues+0x190/0x290
[   24.290281] [<ffffff80081027f4>] hrtimer_interrupt+0xac/0x1bc
[   24.290835] [<ffffff80087881a0>] arch_timer_handler_phys+0x38/0x50
[   24.291423] [<ffffff80080f2560>] handle_percpu_devid_irq+0xcc/0x17c
[   24.292020] [<ffffff80080edd38>] generic_handle_irq+0x2c/0x44
[   24.292569] [<ffffff80080ee0b8>] __handle_domain_irq+0xb4/0xb8
[   24.293117] [<ffffff8008080d70>] gic_handle_irq+0x78/0xc8
```

```
[   24.293629] Exception stack(0xffffffc00a2b3b30 to 0xffffffc00a2b3c60)
[   24.294235] 3b20:                                   0000000000002d2c 0000000029877ce3
[   24.294983] 3b40: 00000000000003e8 00000040f6056000 0000000000000000 0000000000000000
[   24.295729] 3b60: 0000000000000067 ffffff8008f8d3f8 ffffff8008354b3c 636e797320746f6e
[   24.296477] 3b80: 534656203a676e69 656c62616e55203a 6e756f6d206f7420 6620746f6f722074
[   24.297220] 3ba0: 6b6e75206e6f2073 0000000000000000 0000000000000000 000000000000000a
[   24.297967] 3bc0: ffffff808909f857 0000000000005dc0 0000000000000001 0000000000005780
[   24.298715] 3be0: 0000000000005848 0000000000000011 0000000000418958 ffffff8008ca4b9b
[   24.299463] 3c00: fffffffffffffffa 0000000000008001 ffffff8008e7b4c8 ffffffc00a2b3c60
[   24.300210] 3c20: ffffff80083b1e88 ffffffc00a2b3c60 ffffff80083b1e40 0000000080000045
[   24.300954] 3c40: ffffff80083b1e88 ffffffc00a2b3c80 ffffffffffffffff 0000000020000045
[   24.301703] [<ffffff80080827b4>] el1_irq+0xb4/0x140
[   24.302176] [<ffffff80083b1e40>] __delay+0x24/0x48
[   24.302639] [<ffffff80083b1e88>] __const_udelay+0x24/0x2c
[   24.303161] [<ffffff8008160d60>] panic+0x234/0x238
[   24.303623] [<ffffff8008e211d0>] mount_block_root+0x22c/0x298
[   24.304171] [<ffffff8008e213c8>] mount_root+0x70/0x80
[   24.304653] [<ffffff8008e21578>] prepare_namespace+0x1a0/0x1b0
[   24.305205] [<ffffff8008e20dd0>] kernel_init_freeable+0x1e8/0x220
[   24.305782] [<ffffff8008a1edc0>] kernel_init+0x18/0x100
[   24.306288] [<ffffff8008082ef0>] ret_from_fork+0x10/0x20
```

```
81581044  002a0000
81581048  004e0050
8158104c  00300050
81581050  00300043
81581054  00000038
81581058  815b06ad hal!DefaultInitializeProfiling+0x1
8158105c  81599aed hal!DefaultEnableMonitoring+0x1
81581060  81599ad9 hal!DefaultDisableMonitoring+0x1
81581064  81599b31 hal!DefaultSetInterval+0x1
81581068  815aea31 hal!DefaultQueryInformation+0x1
8158106c  81599b09 hal!DefaultOverflowHandler+0x1
81581070  815b0851 hal!DefaultRestartProfiling+0x1
81581074  00000000
81581078  00000000
8158107c  00000000
81581080  815b0495 hal!ArmInitializeProfiling+0x1
81581084  8159fc71 hal!ArmEnableMonitoring+0x1
81581088  8159fbbd hal!ArmDisableMonitoring+0x1
8158108c  8158b3ad hal!ArmSetInterval+0x1
81581090  815af441 hal!ArmQueryInformation+0x1
81581094  8159fcc1 hal!ArmHardwareOverflowHandler+0x1
81581098  815860ad hal!ArmRestartProfiling+0x1
8158109c  815883f5 hal!ArmQueryProcHaltAllowed+0x1
815810a0  8159fee1 hal!ArmPauseProfiling+0x1
```

识别栈上的数据

识别字符串

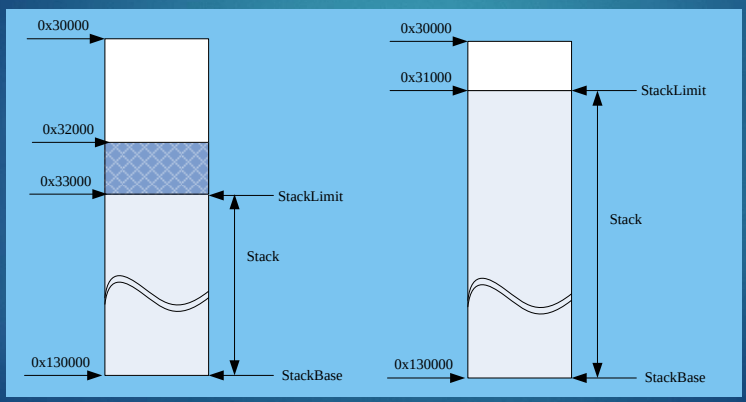
- 根据已知函数原型的识别字符串类型的参数
- 根据地址识别字符串常量
- 使用PEView或者!dh命令寻找.rdata节的起始地址
- 使用db命令浏览

```
SECTION HEADER #2
   .rdata name
    143C virtual size
   22000 virtual address
    2000 size of raw data
   22000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         (no align specified)
         Read Only
```

```
0:000> db 00422000
00422000  00 00 00 00 b0 93 54 49-00 00 00 00 02 00 00 00  ......TI........
00422010  30 00 00 00 00 00 00 00-00 a0 02 00 52 65 61 64  0...........Read
00422020  46 69 6c 65 00 00 00 00-4b 65 72 6e 65 6c 33 32  File....Kernel32
00422030  2e 64 6c 6c 00 00 00 00-48 69 2c 20 65 76 65 72  .dll....Hi, ever
00422040  79 20 6f 6e 65 2e 20 0a-54 68 69 73 20 73 69 6d  y one. .This sim
00422050  70 6c 65 20 75 74 69 6c-69 74 79 20 77 69 6c 6c  ple utility will
00422060  20 73 69 6d 75 6c 61 74-65 20 61 6e 20 61 63 63   simulate an acc
00422070  65 73 73 20 76 69 6f 6c-61 74 69 6f 6e 2e 0a 2d  ess violation..-
```

用户态栈

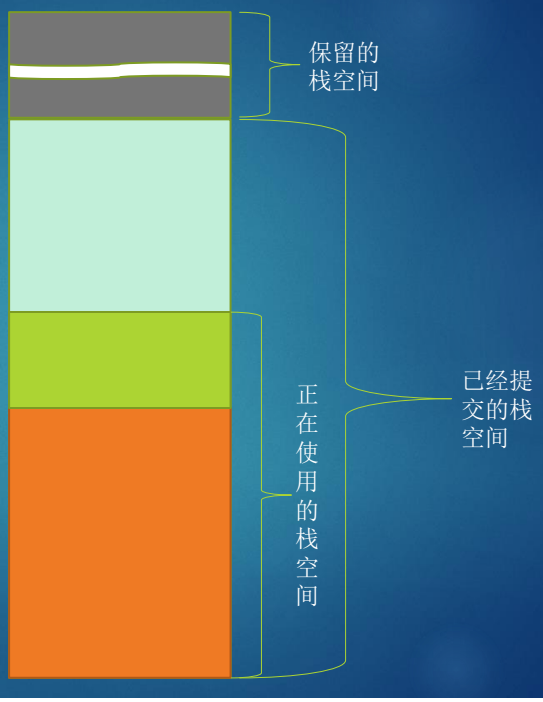
- 用户态栈的缺省保留地址空间是1MB (Linux 8MB)
- 从栈上分配空间意味着栈指针向更低地址移动，腾出更多空间



PTHREAD的线程创建函数

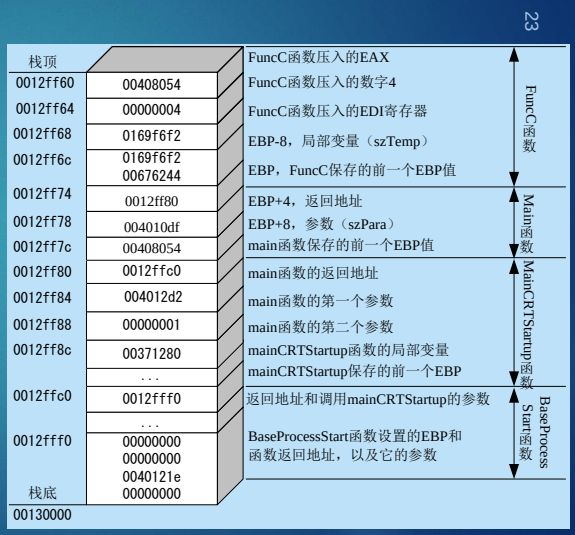
#0 clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:50
#1 0x00007ffff7f9b2ec in create_thread (pd=pd@entry=0x7ffff624c700, attr=attr@entry=0x7fffffffdd40,
stopped_start=stopped_start@entry=0x7fffffffdd3e,
stackaddr=stackaddr@entry=0x7ffff624bfc0,
thread_ran=thread_ran@entry=0x7fffffffdd3f)
at ../sysdeps/unix/sysv/linux/createthread.c:101
#2 0x00007ffff7f9ce10 in __pthread_create_2_1 (newthread=<optimized out>, attr=<optimized out>, start_routine=<optimized out>, arg=<optimized out>) at pthread_create.c:817

栈的典型布局



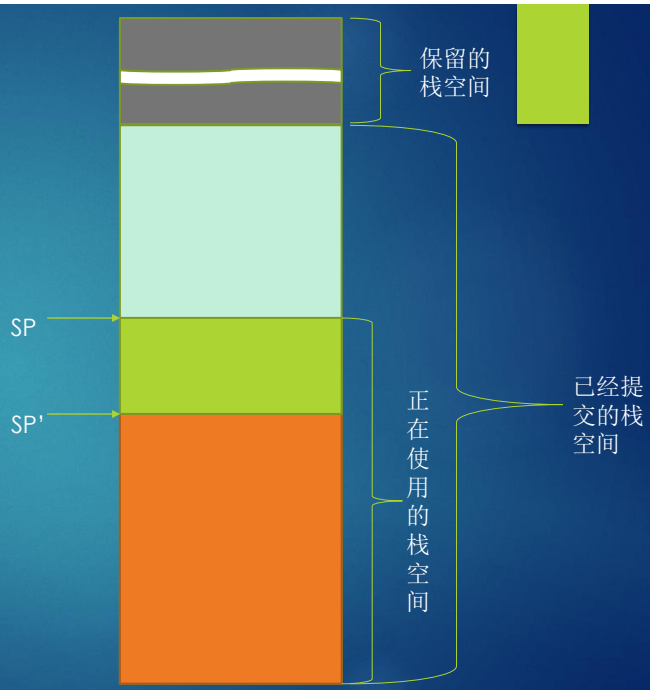
Stack Frame – 栈帧

- 供一个函数使用的一块连续栈区域
- 父函数的在下方
- 子函数的在上方
- 函数返回即释放

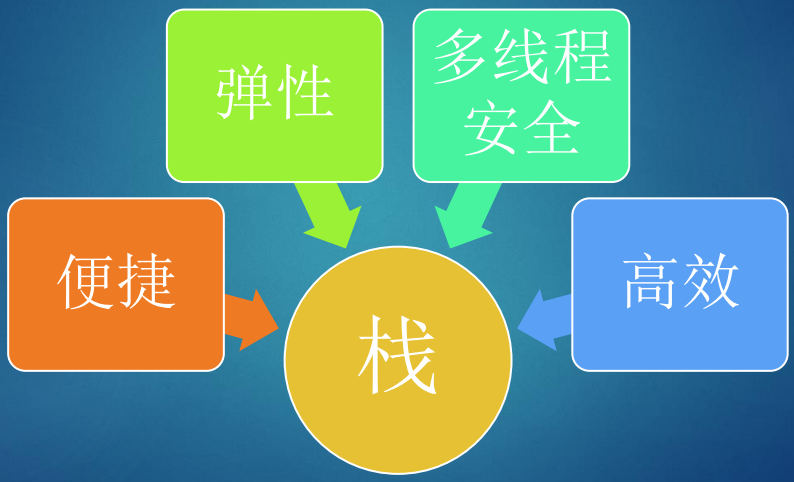


在栈上分配局部变量

- 调整栈指针
  - 做减法
- 释放
  - 做加法



栈的优点



调用子函数

LR[31:1] <== return address
LR[0] <== code type at return address (0 Arm, 1 Thumb)
PC <== subroutine address
...
return address:

LR – Link Register

返回值放入r0，相当于x86的RAX

```
nt!KeGetCurrentStackPointer:
81034e68 4668      mov         r0,sp
81034e6a 4770      bx          lr
```

- 在调用子函数时，ARM 处理器会自动将子函数的返回地址放到这个寄存器中。如果子函数是所谓的叶子函数（不再调用子函数），那么就可以不必额外保存返回地址

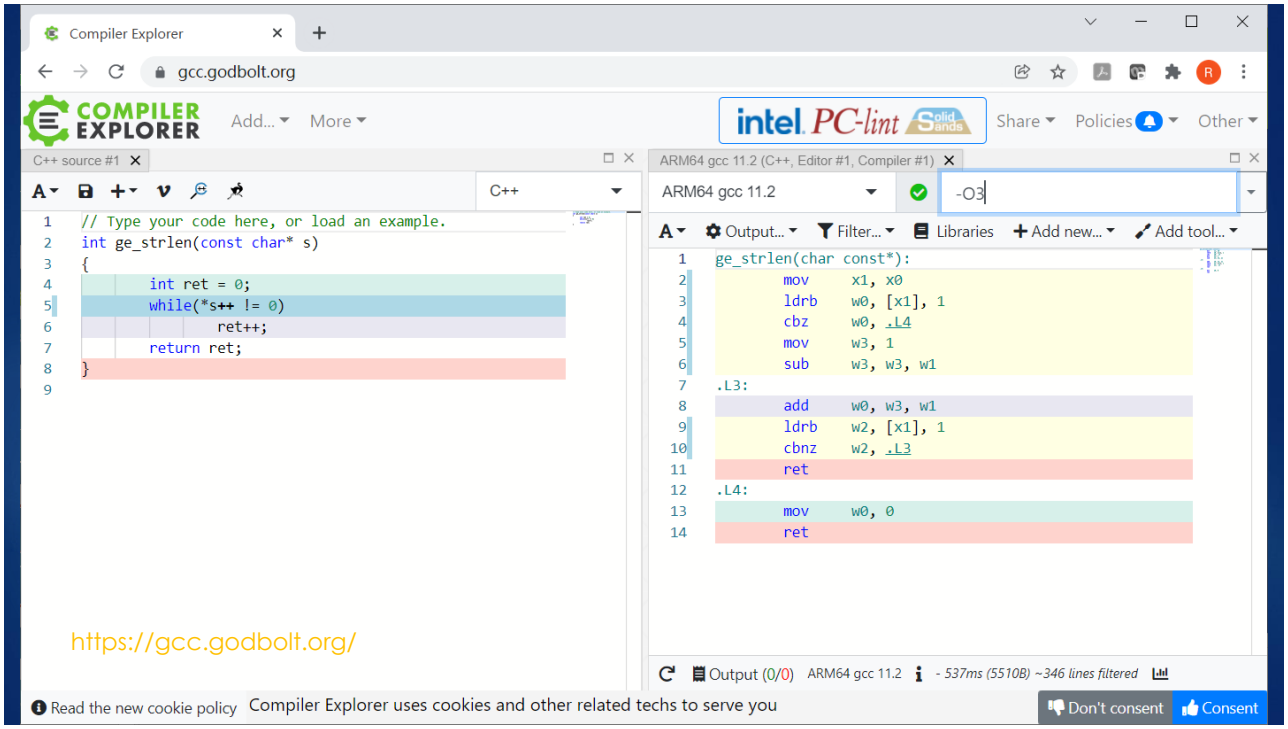
格物

```
5       int ge_strlen(const char* s)
6       {
7               int ret = 0;
8               while(*s++ != 0)
9                       ret++;
10              return ret;
11      }
```

```
Dump of assembler code for function ge_strlen:
   0x0000005555555844 <+0>:     sub     sp, sp, #0x20
   0x0000005555555848 <+4>:     str     x0, [sp, #8]
   0x000000555555584c <+8>:     str     wzr, [sp, #28]
   0x0000005555555850 <+12>:    b       0x5555555860 <ge_strlen+28>
   0x0000005555555854 <+16>:    ldr     w0, [sp, #28]
   0x0000005555555858 <+20>:    add     w0, w0, #0x1
   0x000000555555585c <+24>:    str     w0, [sp, #28]
   0x0000005555555860 <+28>:    ldr     x0, [sp, #8]
   0x0000005555555864 <+32>:    add     x1, x0, #0x1
   0x0000005555555868 <+36>:    str     x1, [sp, #8]
   0x000000555555586c <+40>:    ldrb    w0, [x0]
   0x0000005555555870 <+44>:    cmp     w0, #0x0
   0x0000005555555874 <+48>:    b.ne    0x5555555854 <ge_strlen+16>  // b.any
   0x0000005555555878 <+52>:    ldr     w0, [sp, #28]
   0x000000555555587c <+56>:    add     sp, sp, #0x20
   0x0000005555555880 <+60>:    ret
```

-O2

```
Dump of assembler code for function ge_strlen:
   0x00000000000008b8 <+0>:     add     x1, x0, #0x1
   0x00000000000008bc <+4>:     ldrb    w0, [x0]
   0x00000000000008c0 <+8>:     cbz     w0, 0x8d8 <ge_strlen+32>
   0x00000000000008c4 <+12>:    mov     w0, #0x0                        // #0
   0x00000000000008c8 <+16>:    ldrb    w2, [x1], #1
   0x00000000000008cc <+20>:    add     w0, w0, #0x1
   0x00000000000008d0 <+24>:    cbnz    w2, 0x8c8 <ge_strlen+16>
   0x00000000000008d4 <+28>:    ret
   0x00000000000008d8 <+32>:    mov     w0, #0x0                        // #0
   0x00000000000008dc <+36>:    ret
```



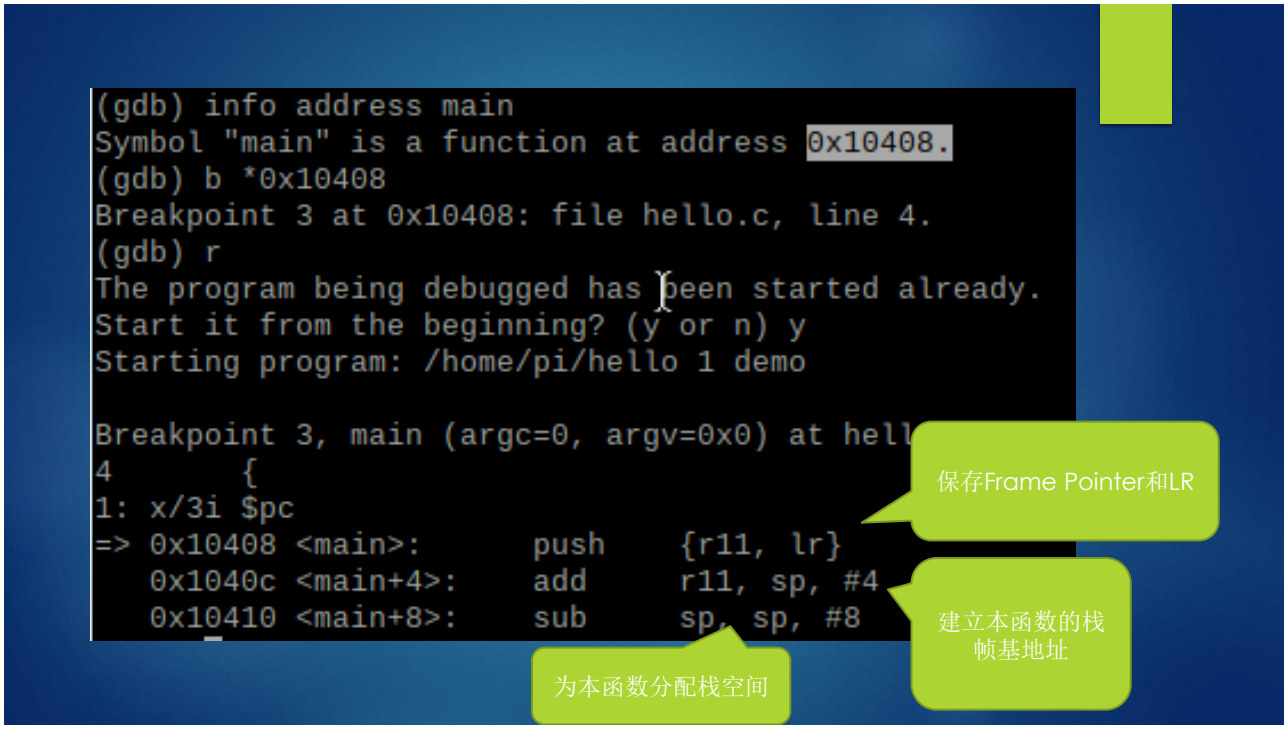
续

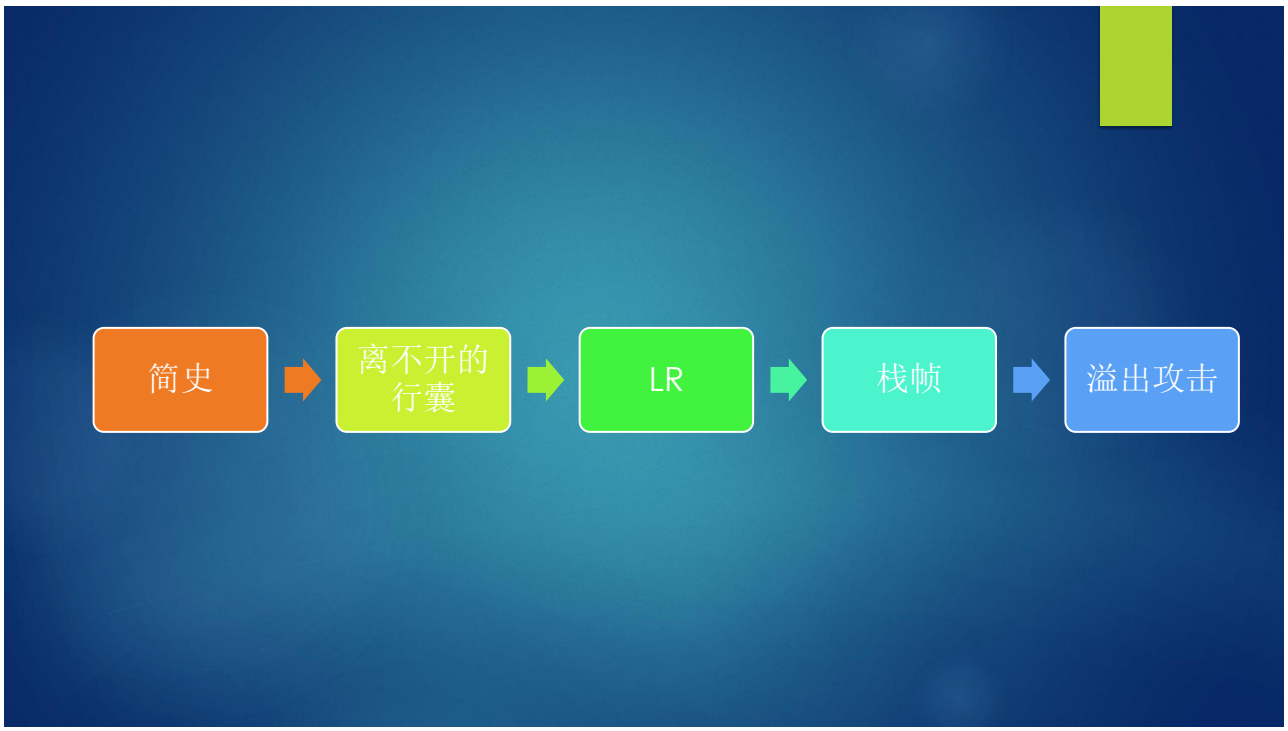
```
nt!KeEnterKernelDebugger:
81152f40 e92d4800 push        {r11,lr}
81152f44 46eb     mov         r11,sp
...
```

保存父函数的栈帧基地址R11和函数返回地址

建立当前函数的栈帧基地址

- 如果是非叶子函数，那么通常在函数开头将LR 的值保存到栈上


(gdb) info address main
Symbol "main" is a function at address 0x10408.
(gdb) b *0x10408
Breakpoint 3 at 0x10408: file hello.c, line 4.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pi/hello 1 demo
Breakpoint 3, main (argc=0, argv=0x0) at hell
4 {
1: x/3i $pc
=> 0x10408 <main>: push {r11, lr}
0x1040c <main+4>: add r11, sp, #4
0x10410 <main+8>: sub sp, sp, #8
保存Frame Pointer和LR
建立本函数的栈帧基地址
为本函数分配栈空间


简史
离不开的行囊
LR
栈帧
溢出攻击

AARCH64

	功能	备注
X0-X28	通用寄存器	可以用W0-W28当32位使用
FP	栈针基地址	也叫X29
LR	Link Register，函数返回地址	也叫X30
SP	栈顶指针	可以用WSP来访问低32位
PC	程序指针	软件不可以直接写
V0-V31	32个SIMD&FP寄存器	用做128位访问时，名叫Q0-Q31，64位时叫D0-D31，32位时叫S0-S31，16位时叫H0-H31，8位时叫B0-B31
PSR	程序状态寄存器	文档里也叫APSR，CPSR，相当于x86的rflags
FPCR	SIMD&FP控制寄存器	
FPSR	SIMD&FP状态寄存器	

A32寄存器

Register	Volatile?	Role
r0	Volatile	Parameter, result, scratch register 1
r1	Volatile	Parameter, result, scratch register 2
r2	Volatile	Parameter, scratch register 3
r3	Volatile	Parameter, scratch register 4
r4	Non-volatile	
r5	Non-volatile	
r6	Non-volatile	
r7	Non-volatile	
r8	Non-volatile	
r9	Non-volatile	
r10	Non-volatile	
r11	Non-volatile	Frame pointer
r12	Volatile	Intra-procedure-call scratch register
r13 (SP)	Non-volatile	Stack pointer
r14 (LR)	Non-volatile	Link register
r15 (PC)	Non-volatile	Program counter

A32寄存器

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8	FP	Frame Pointer or Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable-register 4.
r6	v3		Variable-register 3.
r5	v2		Variable-register 2.
r4	v1		Variable-register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

SB = static base TR = thread register (TR)

```
Reading symbols from ./gearm.out...done.
(gdb) b main
Breakpoint 1 at 0x76c: file /home/geduer/projects/gearm/main.cpp, line 5.
(gdb) r
Starting program: /home/geduer/projects/gearm/bin/ARM64/Debug/gearm.out

Breakpoint 1, Python Exception <class 'NameError'> Installation error: gdb.execute_unwinders function is missing:
main () at /home/geduer/projects/gearm/main.cpp:5
5           printf("%s 向你问好!\n", "gearm");
(gdb) disassemble
Dump of assembler code for function main():
   0x0000005555555764 <+0>:     stp     x29, x30, [sp, #-16]!
   0x0000005555555768 <+4>:     mov     x29, sp
=> 0x000000555555576c <+8>:     adrp    x0, 0x5555555000
   0x0000005555555770 <+12>:    add     x1, x0, #0x830
   0x0000005555555774 <+16>:    adrp    x0, 0x5555555000
   0x0000005555555778 <+20>:    add     x0, x0, #0x838
   0x000000555555577c <+24>:    bl      0x5555555650 <printf@plt>
   0x0000005555555780 <+28>:    mov     w0, #0x0                        // #0
   0x0000005555555784 <+32>:    ldp     x29, x30, [sp], #16
   0x0000005555555788 <+36>:    ret
End of assembler dump.
(gdb) b *0x0000005555555764
Breakpoint 2 at 0x5555555764: file /home/geduer/projects/gearm/main.cpp, line 4.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/geduer/projects/gearm/bin/ARM64/Debug/gearm.out
Python Exception <class 'NameError'> Installation error: gdb.execute_unwinders function is missing:
Python Exception <class 'NameError'> Installation error: gdb.execute_unwinders function is missing:

Breakpoint 2, Python Exception <class 'NameError'> Installation error: gdb.execute_unwinders function is missing:
main () at /home/geduer/projects/gearm/main.cpp:4
4       {
(gdb) disassemble
Dump of assembler code for function main():
=> 0x0000005555555764 <+0>:     stp     x29, x30, [sp, #-16]!
   0x0000005555555768 <+4>:     mov     x29, sp
   0x000000555555576c <+8>:     adrp    x0, 0x5555555000
   0x0000005555555770 <+12>:    add     x1, x0, #0x830
   0x0000005555555774 <+16>:    adrp    x0, 0x5555555000
   0x0000005555555778 <+20>:    add     x0, x0, #0x838
```

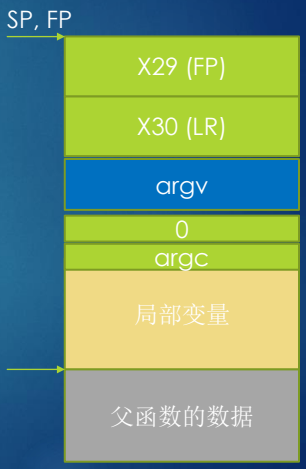
STP – Store Pair Registers

写一对寄存器到内存

```
stp x29, x30, [sp, #-32]!
```

写x29(FP), X30(LR) 到SP-16开始的栈内存

①Ptr = SP-32
②*ptr = X29
③*(ptr+8) = X30
④SP = PTR
⑤PC += 4




```
(gdb) info registers
r0             0x3                 3
r1             0xbefff3e4          3204445156
r2             0xbefff3f4          3204445172
r3             0x10408             66568
r4             0x0                 0
r5             0x10440             66624
r6             0x10318             66328
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0xb6ffd000          3070218240
r11            0x0                 0
r12            0xbefff310          3204444944
sp             0xbefff298          0xbefff298
lr             0xb6e7f718          -1226311912
pc             0x10408             0x10408 <main>
cpsr           0x60000010          1610612752
fpscr          0x0                 0
```

第二个参数　　第一个参数

```
(gdb) x /16x $sp
0xbefff298:     0xb6fb2000      0xbefff3e4      0x00000003      0x00010408
0xbefff2a8:     0x25cd95bb      0x2dd591f7      0x00000000      0x00010440
0xbefff2b8:     0x00010318      0x00000000      0x00000000      0x00000000
0xbefff2c8:     0xb6ffd000      0x00000000      0x00000000      0x00000000
(gdb)
```

```
(gdb) ni
0x0001040c      4       {
   0x00010408 <main+0>: 00 48 2d e9     push    {r11, lr}
=> 0x0001040c <main+4>: 04 b0 8d e2     add     r11, sp, #4
   0x00010410 <main+8>: 08 d0 4d e2     sub     sp, sp, #8
   0x00010414 <main+12>:        08 00 0b e5     str     r0, [r11, #-8]
   0x00010418 <main+16>:        0c 10 0b e5     str     r1, [r11, #-12]
1: x/3i $pc
=> 0x1040c <main+4>:    add     r11, sp, #4
   0x10410 <main+8>:    sub     sp, sp, #8
   0x10414 <main+12>:   str     r0, [r11, #-8]
(gdb) x /16x $sp       r11                lr
0xbefff290:     0x00000000      0xb6e7f718      0xb6fb2000      0xbefff3e4
0xbefff2a0:     0x00000003      0x00010408      0x25cd95bb      0x2dd591f7
0xbefff2b0:     0x00000000      0x00010440      0x00010318      0x00000000
0xbefff2c0:     0x00000000      0x00000000      0xb6ffd000      0x00000000
```

r11 = sp+4

```
(gdb) ni
0x00010410      4       {
   0x00010408 <main+0>: 00 48 2d e9     push    {r11, lr}
   0x0001040c <main+4>: 04 b0 8d e2     add     r11, sp, #4
=> 0x00010410 <main+8>: 08 d0 4d e2     sub     sp, sp, #8
   0x00010414 <main+12>:        08 00 0b e5     str     r0, [r11, #-8]
   0x00010418 <main+16>:        0c 10 0b e5     str     r1, [r11, #-12]
```

```
(gdb) info registers
r0             0x3                 3
r1             0xbefff3e4          3204445156
r2             0xbefff3f4          3204445172
r3             0x10408             66568
r4             0x0                 0
r5             0x10440             66624
r6             0x10318             66328
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0xb6ffd000          3070218240
r11            0xbefff294          3204444820
r12            0xbefff310          3204444944
sp             0xbefff290          0xbefff290
lr             0xb6e7f718          -1226311912
pc             0x10410             0x10410 <main+8>
cpsr           0x60000010          1610612752
fpscr          0x0                 0
```

R11 – frame pointer
组成链条
栈回溯时用来遍历栈帧

```
(gdb) x /16x $r11
0xbefff294:     0xb6e7f718      0xb6fb2000      0xbefff3e4      0x00000003
0xbefff2a4:     0x00010408      0x25cd95bb      0x2dd591f7      0x00000000
0xbefff2b4:     0x00010440      0x00010318      0x00000000      0x00000000
0xbefff2c4:     0x00000000      0xb6ffd000      0x00000000      0x00000000
```

sp = sp - 8

```
(gdb) ni
0x00010414      4       {
   0x00010408 <main+0>: 00 48 2d e9     push    {r11, lr}
   0x0001040c <main+4>: 04 b0 8d e2     add     r11, sp, #4
   0x00010410 <main+8>: 08 d0 4d e2     sub     sp, sp, #8
=> 0x00010414 <main+12>:        08 00 0b e5     str     r0, [r11, #-8]
   0x00010418 <main+16>:        0c 10 0b e5     str     r1, [r11, #-12]
```

从栈上分配8个字节的空间
准备给调用printf函数时传递参数用

新分配好的8字节空间

```
(gdb) x /16x $sp                                        r11             lr
0xbefff288:     0x00000000      0x00000000      0x00000000      0xb6e7f718
0xbefff298:     0xb6fb2000      0xbefff3e4      0x00000003      0x00010408
0xbefff2a8:     0x25cd95bb      0x2dd591f7      0x00000000      0x00010440
0xbefff2b8:     0x00010318      0x00000000      0x00000000      0x00000000
```

```
0x00010414 <main+12>:        08 00 0b e5     str     r0, [r11, #-8]
0x00010418 <main+16>:        0c 10 0b e5     str     r1, [r11, #-12]
```

```
(gdb) x /16x $sp
0xbefff288:     0xbefff3e4      0x00000003      0x00000000      0xb6e7f718
0xbefff298:     0xb6fb2000      0xbefff3e4      0x00000003      0x00010408
0xbefff2a8:     0x25cd95bb      0x2dd591f7      0x00000000      0x00010440
0xbefff2b8:     0x00010318      0x00000000      0x00000000      0x00000000
```

```
5            printf("hello rpi %d %s\n",argc,argv);
1: x/3i $pc
=> 0x1041c <main+20>:    ldr     r2, [r11, #-12]
   0x10420 <main+24>:    ldr     r1, [r11, #-8]
   0x10424 <main+28>:    ldr     r0, [pc, #16]   ; 0x1043c <main+52>
```

argv

argc

```
(gdb) p *(char**)0x1043c
$3 = 0x104b0 "hello rpi %d %s\n"
```

字符串常量，使用PC来计算地址

```
   0x00010408 <+0>:     push    {r11, lr}
   0x0001040c <+4>:     add     r11, sp, #4
   0x00010410 <+8>:     sub     sp, sp, #8
   0x00010414 <+12>:    str     r0, [r11, #-8]
   0x00010418 <+16>:    str     r1, [r11, #-12]
   0x0001041c <+20>:    ldr     r2, [r11, #-12]
   0x00010420 <+24>:    ldr     r1, [r11, #-8]
   0x00010424 <+28>:    ldr     r0, [pc, #16]   ; 0x1043c <main+52>
=> 0x00010428 <+32>:    bl      0x102e8 <printf@plt>
   0x0001042c <+36>:    mov     r3, #8
   0x00010430 <+40>:    mov     r0, r3
   0x00010434 <+44>:    sub     sp, r11, #4
   0x00010438 <+48>:    pop     {r11, pc}
   0x0001043c <+52>:                    ; <UNDEFINED> instruction: 0x
```

Branch with Link

参数

```
(gdb) info registers
r0             0x104b0             66736
r1             0x3                 3
r2             0xbefff3e4          3204445156
r3             0x10408             66568
r4             0x0                 0
r5             0x10440             66624
r6             0x10318             66328
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0xb6ffd000          3070218240
r11            0xbefff294          3204444820
r12            0xbefff310          3204444944
sp             0xbefff288          0xbefff288
lr             0xb6e7f718          -1226311912
pc             0x10428             0x10428 <main+32>
cpsr           0x60000010          1610612752
fpscr          0x0                 0
```

Procedure Linkage Table

```
0x000102e8 in printf@plt ()
1: x/3i $pc
=> 0x102e8 <printf@plt>:        add     r12, pc, #0, 12
   0x102ec <printf@plt+4>:      add     r12, r12, #16, 20       ; 0x10000
   0x102f0 <printf@plt+8>:      ldr     pc, [r12, #3356]!       ; 0xd1c
```

Hello程序的导入函数表

```
(gdb) x /16w $r11-4
0xbefff260:     0xbefff294      0xb6eb0470      0xbefff27c      0x74c4e400
0xbefff270:     0x00000000      0x0001042c      0x000104b0      0x00000003
0xbefff280:     0xbefff3e4      0x00010408      0xbefff3e4      0x00000003
0xbefff290:     0x00000000      0xb6e7f718      0xb6fb2000      0xbefff3e4
(gdb) bt
#0  0xb6ea9108 in _IO_vfprintf_internal (s=0xb6fb2d90 <_IO_2_1_stdout_>,
    format=0x104b0 "hello rpi %d %s\n", format@entry=0x0, ap=...,
    ap@entry=...) at vfprintf.c:1239
#1  0xb6eb0470 in __printf (format=0x104b0 "hello rpi %d %s\n")
    at printf.c:33
#2  0x0001042c in main (argc=3,
    argv=0xbefff3e4 "S\365\377\276b\365\377\276d\365\377\276") at hello.c:5
```

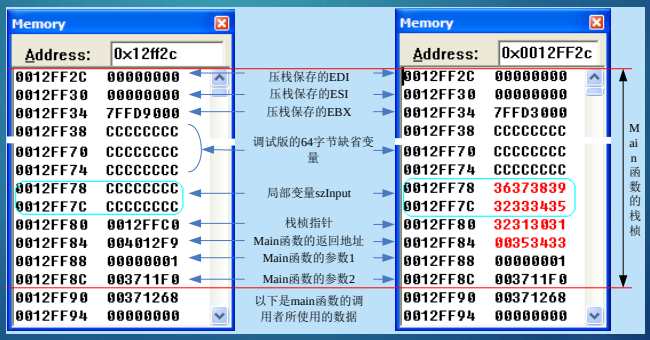
返回值

整数	x0	

https://docs.microsoft.com/en-us/cpp/build/arm64-windows-abi-conventions?view=msvc-160

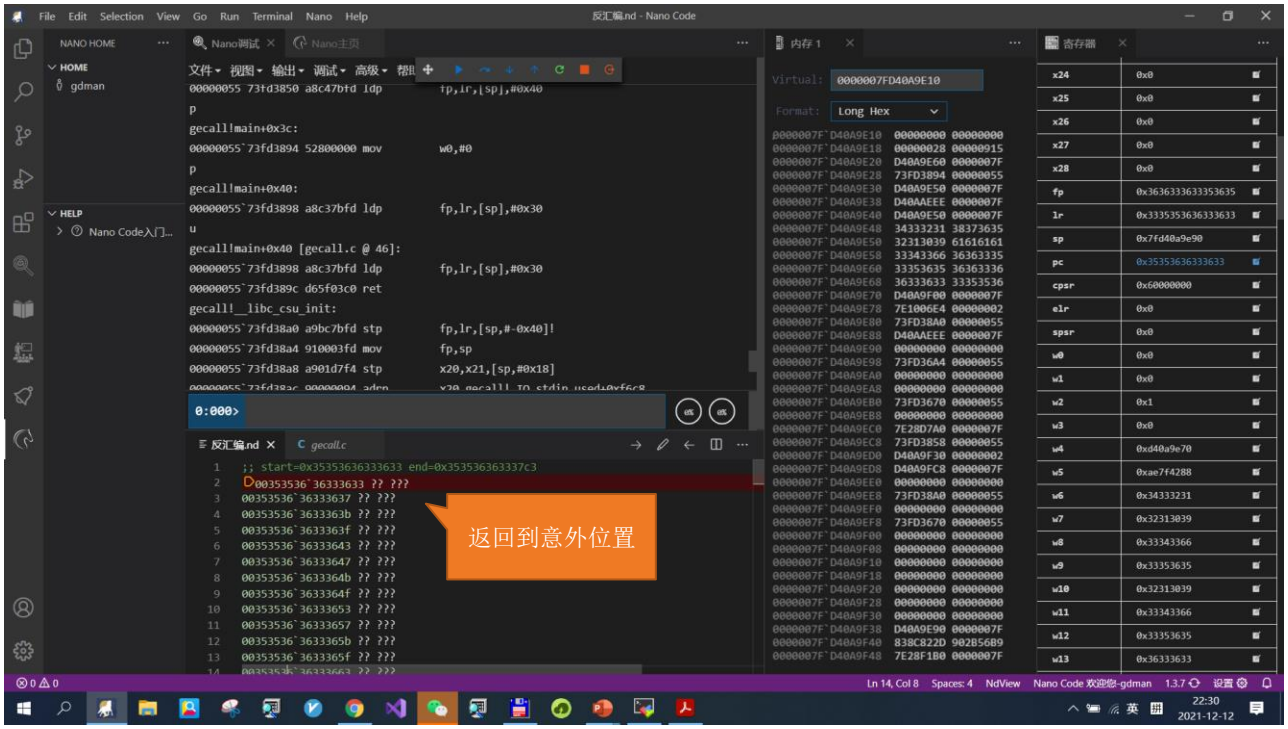
缓冲区溢出

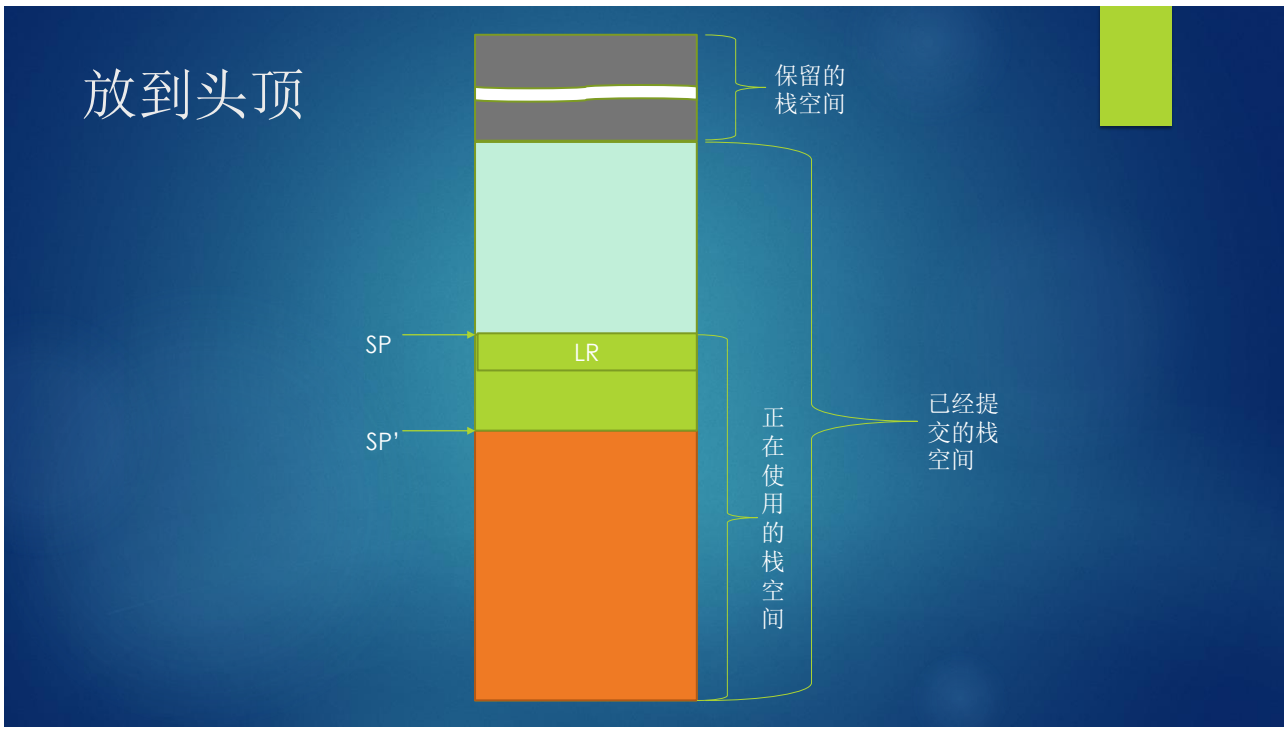
- ARM架构中，栈是向低地址方向生长的
- 向缓冲区写入超过其容量的内容可以覆盖栈上的函数返回地址，使函数返回到意外的地方
- 栈溢出攻击

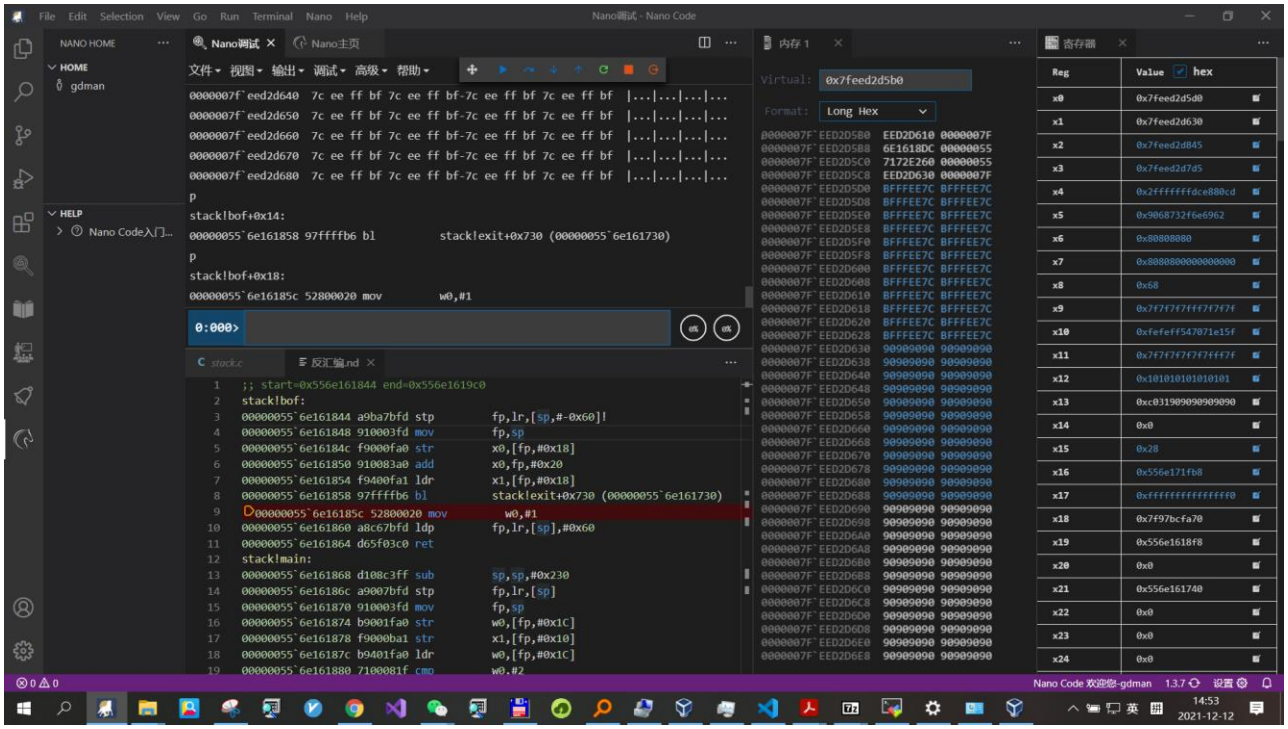


demo

缓冲区溢出演示


返回到意外位置


放到头顶
保留的
栈空间
SP
LR
SP'
正在使用的栈空间
已经提交的栈空间



Return Oriented Program

```
R9 : 0x7ffff7fe0d50 (endbr64)
R10: 0x0
R11: 0x0
R12: 0x401070 (<_start>:        endbr64)
R13: 0x7fffffffdf80 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x401178 <vuln+34>:  nop
   0x401179 <vuln+35>:  leave
   0x40117a <vuln+36>:  ret
=> 0x40117b <main>:     endbr64
   0x40117f <main+4>:   push   rbp
   0x401180 <main+5>:   mov    rbp,rsp
   0x401183 <main+8>:   mov    eax,0x0
   0x401188 <main+13>:  call   0x401156 <vuln>
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffde98 --> 0x7ffff7ded0b3 (<__libc_start_main+243>:       mov    edi,eax)
0008| 0x7fffffffdea0 --> 0x7ffff7ffc620 --> 0x50b1600000000
0016| 0x7fffffffdea8 --> 0x7fffffffdf88 --> 0x7fffffffe2d1 ("/home/gebox/ret2libc/cns/labs/08-rop/00-tutorial-2-ret2li
bc/ret2libc")
0024| 0x7fffffffdeb0 --> 0x100000000
0032| 0x7fffffffdeb8 --> 0x40117b (<main>:      endbr64)
0040| 0x7fffffffdec0 --> 0x4011b0 (<__libc_csu_init>:   endbr64)
0048| 0x7fffffffdec8 --> 0xcb88f43ecc4added
0056| 0x7fffffffded0 --> 0x401070 (<_start>:    endbr64)
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, main () at ret2libc.c:10
10      int main() {
gdb-peda$ p system
$1 = {int (const char *)} 0x7ffff7e1b410 <__libc_system>
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0x7ffff7f7d5aa --> 0x68732f6e69622f ('/bin/sh')
```

Breakpoint 1, main () at ret2libc.c:10
10 int main() {
gdb-peda$ p system

$1 = {int (const char *)} **0x7ffff7e1b410** <__libc_system>
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:

libc : **0x7ffff7f7d5aa** --> 0x68732f6e69622f ('/bin/sh')

Breakpoint 1, main () at ret2libc.c:10
10 int main() {
gdb-peda$ p system

$1 = {int (const char *)} **0x7ffff7e1b410** <__libc_system>
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:

libc : **0x7ffff7f7d5aa** --> 0x68732f6e69622f ('/bin/sh')

```
gebox@gebox-VirtualBox:~/ret2libc/cns/labs/08-rop/00-tutorial-2-ret2libc$ python3 exploit.py
[*] '/home/gebox/ret2libc/cns/labs/08-rop/00-tutorial-2-ret2libc/ret2libc'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process './ret2libc': pid 17434
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$ ls
[*] Process './ret2libc' stopped with exit code -11 (SIGSEGV) (pid 17434)
[*] Got EOF while sending in interactive
Traceback (most recent call last):
  File "/home/gebox/.local/lib/python3.8/site-packages/pwnlib/tubes/process.py", line 787, in close
    fd.close()
BrokenPipeError: [Errno 32] Broken pipe
```

```
gdb-peda$ dumprop
Warning: this can be very slow, do not run for large memory range
Writing ROP gadgets to file: ret2libc-rop.txt ...
0x40113e: ret
0x4010a3: cli; ret
0x4010cf: nop; ret
0x401179: leave; ret
0x4010a1: nop edx; ret
0x4010a0: endbr64; ret
0x40110e: jmp rax; ret
0x40113d: pop rbp; ret
0x401213: pop rdi; ret
0x401212: pop r15; ret
0x4010ce: xchg ax,ax; ret
0x401178: nop; leave; ret
0x401017: add esp,0x8; ret
0x401016: add rsp,0x8; ret
0x40109f: nop; endbr64; ret
0x401211: pop rsi; pop r15; ret
0x401210: pop r14; pop r15; ret
0x40109e: hlt; nop; endbr64; ret
0x4011fc: fisttp [rax-0x7d]; ret
0x40121f: add bl,dh; nop edx; ret
0x4010cc: jmp rax; xchg ax,ax; ret
0x4011a1: mov eax,0x0; pop rbp; ret
0x401014: call rax; add rsp,0x8; ret
0x40113b: add [rcx],al; pop rbp; ret
0x4011a4: add [rax],al; pop rbp; ret
--More--(25/116)
0x40121e: add [rax],al; endbr64; ret
0x40109d: add ah,dh; nop; endbr64; ret
0x40113c: add [rbp-0x3d],ebx; nop; ret
0x40122d: sub esp,0x8; add rsp,0x8; ret
0x40122c: sub rsp,0x8; add rsp,0x8; ret
```

$ apt-get update
$ apt-get install python3 python3-pip python3-dev git libssl-dev libffi-dev build-essential
$ python3 -m pip install --upgrade pip
$ python3 -m pip install --upgrade pwntools

```
gdb-peda$ x /10gx 0x7fffffffde90
0x7fffffffde90: 0x0000000000000000      0x00007ffff7ded0b3
0x7fffffffdea0: 0x00007ffff7ffc620      0x00007fffffffdf88
0x7fffffffdeb0: 0x0000000100000000      0x000000000040117b
0x7fffffffdec0: 0x00000000004011b0      0x75697a163a7af78a
0x7fffffffded0: 0x0000000000401070      0x00007fffffffdf80
gdb-peda$ info symbol 0x000000000040117b
main in section .text of /home/gebox/ret2libc/cns/labs/08-rop/00-tutorial-2-ret2libc/ret2libc
```

保护

-fno-stack-protector

检测到溢出

```
Child-SP          RetAddr           Call Site
0000007f`ebd48dc0 0000007f`953d18d4 libc_so!raise+0xb0
0000007f`ebd48dc0 0000007f`9540a68c libc_so!abort+0x154
0000007f`ebd48dc0 0000007f`9547f284 libc_so!__fsetlocking+0x2f4
0000007f`ebd48dc0 0000007f`9547f2bc libc_so!__stack_chk_fail+0x64
0000007f`ebd48dc0 0000007f`9547cfa4 libc_so!__fortify_fail+0x14
0000007f`ebd48dc0 0000007f`9547c0ec libc_so!__chk_fail+0x14
0000007f`ebd48dc0 00000055`62f7e77c libc_so!__strcpy_chk+0x4c
0000007f`ebd48dc0 0000007f`953be720 gecall!main(void)+0x3c
0000007f`ebd48dc0 00000055`62f7e7e8 libc_so!__libc_start_main+0xe0
0000007f`ebd48dc0 00000000`00000000 gecall!_entry+0x34
```

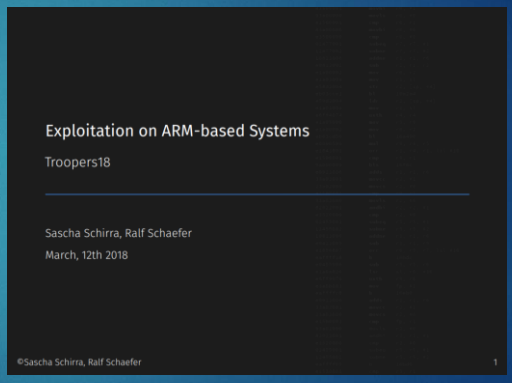
Name	Description
CVE-2021-43579	A stack-based buffer overflow in image_load_bmp() in HTMLDOC before 1.9.13 results in remote code execution if the victim converts an HTML document linking to a crafted BMP file.
CVE-2021-42012	A stack-based buffer overflow vulnerability in Trend Micro Apex One, Apex One as a Service and Worry-Free Business Security 10.0 SP1 could allow a local attacker to escalate privileges on affected installations. Please note: an attacker must first obtain the ability to execute low-privileged code on the target system in order to exploit this vulnerability.
CVE-2021-41794	ogs_fqdn_parse in Open5GS 1.0.0 through 2.3.3 inappropriately trusts a client-supplied length value, leading to a buffer overflow. The attacker can send a PFCP Session Establishment Request with "internet" as the PDI Network Instance. The first character is interpreted as a length value to be used in a memcpy call. The destination buffer is only 100 bytes long on the stack. Then, 'i' gets interpreted as 105 bytes to copy from the source buffer to the destination buffer.
CVE-2021-41459	There is a stack buffer overflow in MP4Box v1.0.1 at src/filters/dmx_nhml.c:1008 in the nhmldmx_send_sample() function szXmlFrom parameter which leads to a denial of service vulnerability.
CVE-2021-41457	There is a stack buffer overflow in MP4Box 1.1.0 at src/filters/dmx_nhml.c in nhmldmx_init_parsing which leads to a denial of service vulnerability.
CVE-2021-41456	There is a stack buffer overflow in MP4Box v1.0.1 at src/filters/dmx_nhml.c:1004 in the nhmldmx_send_sample() function szXmlTo parameter which leads to a denial of service vulnerability.
CVE-2021-39847	XMP Toolkit SDK version 2020.1 (and earlier) is affected by a stack-based buffer overflow vulnerability potentially resulting in arbitrary code execution in the context of the current user. Exploitation requires user interaction in that a victim must open a crafted file.
CVE-2021-39595	An issue was discovered in swftools through 20200710. A stack-buffer-overflow exists in the function rfx_alloc() located in mem.c. It allows an attacker to cause code Execution.
CVE-2021-39561	An issue was discovered in swftools through 20200710. A stack-buffer-overflow exists in the function Gfx::opSetFillColorN() located in Gfx.cc. It allows an attacker to cause code Execution.
CVE-2021-39558	An issue was discovered in swftools through 20200710. A stack-buffer-overflow exists in the function VectorGraphicOutputDev::drawGeneralImage() located in VectorGraphicOutputDev.cc. It allows an attacker to cause code Execution.
CVE-2021-39540	An issue was discovered in pdftools through 20200714. A stack-buffer-overflow exists in the function Analyze::AnalyzePages() located in analyze.cpp. It allows an attacker to cause code Execution.
CVE-2021-39531	An issue was discovered in libslax through v0.22.1. slaxLexer() in slaxlexer.c has a stack-based buffer overflow.
CVE-2021-3928	vim is vulnerable to Stack-based Buffer Overflow
CVE-2021-38684	A stack buffer overflow vulnerability has been reported to affect QNAP NAS running Multimedia Console. If exploited, this vulnerability allows attackers to execute arbitrary code. We have already fixed this vulnerability in the following versions of Multimedia Console: Multimedia Console 1.4.3 (2021/10/05) and later
CVE-2021-38525	Certain NETGEAR devices are affected by a stack-based buffer overflow by an authenticated user. This affects D3600 before 1.0.0.76, D6000 before 1.0.0.76, D6200 before 1.1.00.36, D7000 before 1.0.1.70, EX6200v2 before 1.0.1.78, EX7000 before 1.0.1.78, EX8000 before 1.0.1.186, JR6150 before 1.0.1.18, PR2000 before 1.0.0.28, R6020 before 1.0.0.42, R6050 before 1.0.1.18, R6080 before 1.0.0.42, R6120 before 1.0.0.46, R6220 before 1.1.0.80, R6260 before 1.1.0.64, R6300v2 before 1.0.4.34, R6700 before 1.0.2.6, R6700v2 before 1.2.0.36, R6800 before 1.2.0.36, R6900 before 1.0.2.4, R6900P before 1.3.1.64, R6900v2 before 1.2.0.36, R7000 before 1.0.9.42, R7000P before 1.3.1.64, R7800 before 1.0.2.60, R8900 before 1.0.4.12, R9000 before 1.0.4.12, and XR500 before 2.3.2.40.

http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=stack+buffer+overflow

软件安全的核心问题是代码与数据的混乱，数据摇身变为代码。

资源

- https://raw.githubusercontent.com/sashs/arm_exploitation/master/exploitation_on_arm_based_systems.pdf


Exploitation on ARM-based Systems

Troopers18

Sascha Schirra, Ralf Schaefer

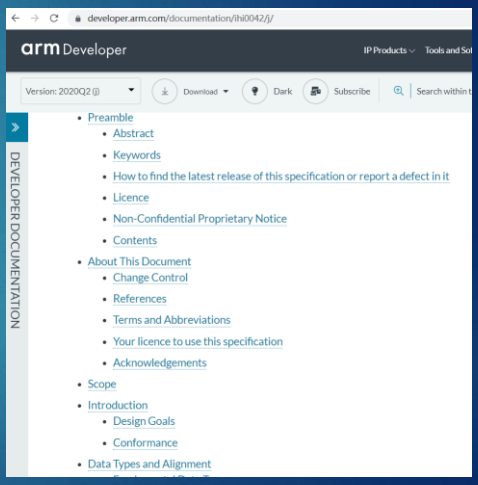
March, 12th 2018

©Sascha Schirra, Ralf Schaefer 1

ABI

- **Procedure Call Standard for the Arm Architecture - ABI 2020Q2 documentation**
- Procedure Call Standard for the Arm® Architecture

https://developer.arm.com/documentation/ihi0042/j/



切问而近思

欢迎关注格友公众号

```
$ git clone https://github.com/longld/peda.git ~/peda
$ echo “source ~/peda/peda.py” >> ~/.gdbinit
```