==Phrack Inc.==

Volume 0x0c, Issue 0x40, Phile #0x0f of 0x11

|=---=|

|=-------------=[Blind TCP/IP hijacking is still alive]=---------------=|

|=---=|

|=---=|

|=--------------------=[By lkm]=-----------------------=|

|=-------------------=[<lkm_at_phrack_dot_org>]=-----------------------=|

|=---=|

편역: poc@securityproof.net

번역 과정에서 오역 및 오타가 발생할 수 있습니다. 엄밀한 글읽기를 원하시는 분은 원 텍스트를 읽으시기 바랍니다. 참고로 원문에 정확하지 않은 영문들이 사용되고 있는 부분들이 있습니다. 저자가 native speaker인지는 확인을 하지 않았습니다. 가장 좋은 소스는 원문 텍스트입니다.

--[Contents

—[1 - 도입

TCP를 가지고 노는 것(blind spoofing/hijacking, 등...)은 initials TCP sequence number(ISN)를 추측 가능 했던(64K 룰, 등...) 몇 년 전에는 아주 인기가 있었다. 요즘은 ISN들이 랜덤화가 아주 잘 되어 있어 이제는 공격이 거의 불가능한 것처럼 보인다.

이 글에서는 blind TCP hijacking 공격을 수행하는 것이 [1][1]에서처럼 ISN을 생성하는 역할을 하는 PRNG를 공격하지 않고도 요즘 여전히 가능하다는 것을 보여줄 것이다. 다음 시스템((Windows 2K, windows XP, 그리고 FreeBSD 4)들에 적용될 수 있는 방법도 제시할 것이다. 이 방법은 구현하기에 직관적이지는 않지만 그럼에도 불구하고 완전히 가능하며, 모든 취약한 시스템에 대해 이 공격을 수행하는데 성공적으로 사용된 툴을 코딩했기 때문이다.

—[2 - 필수 전제

이 섹션에서는 이 글 전체를 이해하는데 필요한 정보를 몇 가지 제시할 것이다.

——[2.1 - TCP에 대한 간단한 상기

두 개의 호스트(이 글에서는 앞으로 각각 "client"와 "server"라고 지칭할 것임) 사이에 TCP 연결은 client-IP, server-IP, client-port, server-port에 의해 확인될 수 있다. 서버 포트가 'well known' 포트라면 클라이언트의 포트는 보통 1024-5000 범위일 것이며, 운영체제에 의해 자동으로 부여된다.(예: 어떤 사람으로부터 freenode로의 연결은 [ppp289.someISP.com, irc.freenode.net, 1207, 6667]에 의해 나타날 수 있다.)

TCP 연결 상에서 통신이 이루어지면 교환된 TCP 패킷 헤더들은 이 정보들(실제로, IP 헤더는 출발지/목적지 IP를 가지고 있으며, 그리고 TCP 헤더는 출발지/목적지 포트를 가지고 있다)을 가지고 있다. 각 TCP 패킷 헤더는 또한 sequence number(SEQ) 및 acknowledgement number(ACK)에 대한 필드도 가지고 있다.

연결에 포함된 이 두 호스트의 각각은 연결이 확립될 때 무작위로 32비트 SEQ number를 연산한다. 이 초기 initial SEQ number는 ISN이라고 부른다. 그런 다음, 한 호스트가 N 바이트의 데이터로 패킷을 보낼 때마다 SEQ 번호에 N을 더한다. sender는 외부로 나가는 각 TCP 패킷의 SEQ 필드에 그의 현재 SEQ를 더한다. ACK 필드는 다른 호스트로부터 다음 예상 SEQ number로 가득 찬다. 각 호스트들은 그 자신의 다음 sequence number(SND.NEXT)를 유지하고, 그리고 다른 호스트(RCV.NEXT)로부터 다음 예상 SEQ number를 유지한다.

한가지 예를 들어 확인해보자.(간단히 하기 위해, 이 연결이 이미 확립되고, 포트들이 보이지 않는다고 생각해보자.)

---

[1] 역자 주: "Strange Attractors and TCP/IP Sequence Number Analysis - One Year Later" (http://lcamtuf.coredump.cx/newtcp/)

[==]

Client Server

[SND.NEXT=1000] [SND.NEXT=2000]

--[SEQ=1000, ACK=2000, size=20]->

[SND.NEXT=1020] [SND.NEXT=2000]

<-[SEQ=2000, ACK=1020, size=50]--

[SND.NEXT=1020] [SND.NEXT=2050]

--[SEQ=1020, ACK=2050, size=0]->

[==]

위의 예에서, 먼저 클라이언트는 20바이트의 데이터를 보낸다. 그런 다음, 서버는 이 데이터를 승인하고(ACK=1020), 같은 패킷에 자신의 50바이트의 데이터를 보낸다. 클라이언트가 보낸 마지막 패킷은 우리가 "simple ACK"라고 부를 것이다. 그것은 서버가 보낸 50바이트의 데이터를 승인하지만 어떤 데이터도 탑재하여(data payload) 운반하지는 않는다. "simple ACK"는 호스트가 받은 데이터를 승인하지만 아직 전송할 데이터를 가지고 있지 않은 곳에서 사용된다. 분명히, 어떤 잘 형성된 "simple ACK" 패킷은 무한 루프로 연결될 것이기 때문에, 승인되지 않을 것이다. 개념적으로, 각 바이트는 sequence number를 가지고 있으며, 그것은 TCP 헤더 필드에 포함된 SEQ는 첫 바이트의 sequence number를 나타낸다. 예를 들어, 첫 패킷의 20 바이트는 sequence number 1000..1019를 가진다.

TCP는 "window"의 개념을 정의함으로써 흐름 통제(flow control) 메커니즘을 구현한다. 각 호스트는 우리가 RCV.WND라고 부를 TCP window 사이즈(각 TCP 연결에 따라 동적이고 특수하며, TCP 패킷에 선정되어 있음)를 가지고 있다.

언제라도 하나의 호스트는 RCV.NXT와 (RCV.NXT+RCV.WND-1) 사이에 sequence number를 가진 바이트를 받아들일 것이다. 이 메커니즘은 그 호스트에 "전송 중인"(in transit) RCV.WND 바이트보다 더 많을 수 없다는 것을 언제라도 확인해준다.

연결의 확립과 해체(teardown)는 TCP 헤더의 플래그에 의해 관리된다. 이 글에서 유일하게 유용한 플래그들은 SYN, ACK, 그리고 RST(추가 정보에 대해서는 RFC793를 참고할 것)이다. SYN와 ACK 플래그는 연결 확립에서 다음과 같이 사용된다:

[==]

Client Server

[client picks an ISN]

[SND.NEXT=5000]

--[flags=SYN, SEQ=5000]--> [server picks an ISN]

[SND.NEXT=5001] [SND.NEXT=9000]

<-[flags=SYN+ACK, SEQ=9000, ACK=5001]--

[SND.NEXT=5001] [SND.NEXT=9001]

--[flags=ACK, SEQ=5001, ACK=9001]-->

...connection established...

[==========]

이 확립 동안 각 호스트들의 SND.NEXT는 1씩 증가한다고 언급할 것이다. 그것은 sequence number가 관련되어 있는 한 SYN 플래그가 (가상의) 1 바이트로 카운트되기 때문이다. 그래서, SYN 플래그 세트를 가진 어떤 패킷은 1+packet_data_size(여기서 데이터 사이즈는 0임)씩 SND.NEXT를 증가시킨다. 독자들은 또한 ACK 필드는 옵션이라는 것을 목격할 것이다. ACK 필드는 비록 관련은 있지만 ACK 플래그와 혼동해서는 안된다. ACK 플래그는 ACK 필드가 존재하면 설정된다. ACK 플래그는 확립된 연결에 속한 패킷에만 항상 설정된다.

RST 플래그는 (닫힌 포트에 연결을 시도하는 것과 같은 에러 때문에 발생하는) 연결을 비정상적으로 닫기 위해 사용된다.

─── [2.2 - IP ID의 흥미로운 부분

IP 헤더는 IP fragmentation/reassembly 메커니즘에 의해 사용되는 16비트 정수인 IP_ID라는 플래그를 가지고 있다. 이 수는 어떤 한 호스트에 의해 보내진 각 IP 패킷을 유일한 것으로 나타내기 위해 필요하지만 fragmentation에 의해 변경되지 않을 것이다(그래서, 같은 패킷의 프래그먼트들은 같은 IP ID를 가질 것이다.).

이제, 독자들은 IP_ID가 왜 흥미로운지 궁금해할 것이다. 몇몇 TCP/IP 스택(Windows 98, 2K, 및 XP를 포함. 이 스택들은 전역 변수 counter에 IP_ID를 저장하고 있으며, 그것은 각 보내진 IP 패킷으로 단순하게 증가한다.)에는 멋진 "기능"을 있다. 이것은 공격자가 어떤 한 호스트의 IP_ID 카운트를 확인할 수 있게 해주며(예를 들어, ping으로), 그래서 그 호스트가 패킷을 보낼 때를 안다.

예:

[==========]

attacker Host

```
                    --[PING]->
          <-[PING REPLY, IP_ID=1000]--

... wait a little ...

                    --[PING]->
          <-[PING REPLY, IP_ID=1010]--

<attacker> Uh oh, the Host sent 9 IP packets between my pings.
[=============================================================================]
```

이 테크닉은 잘 알려져 있으며, 이미 스텔스 포트스캔([3][2]와 [5][3])를 실제로 수행하기 위해 사용되어 왔다.

——[2.3 - 수집할 정보의 목록

자, 기존의 TCP 연결을 하이재킹하기 위해 무엇이 필요한가?

먼저, 우리는 클라이언트의 IP, 서버의 IP, 클라이언트 포트, 서버의 포트를 알 필요가 있다. 이 글에서 클라이언트의 IP, 서버의 IP, 서버의 포트는 알고 있는 것으로 가정할 것이다. 어려운 것은 클라이언트의 포트를 탐지하는 것인데, 왜냐하면 클라이언트의 포트는 OS에 의해 무작위로 부여되기 때문이다. 다음 섹션에서 우리는 IP_ID로 어떻게 클라이언트의 포트를 알아낼 것인지 살펴볼 것이다.

우리가 두 가지 방법(서버로부터 클라이언트에 데이터를 보내고, 서버로부터 클라이언트에 데이터를 보내는 것)을 하이재킹할 수 있기를 원한다면 필요한 다음 것은 서버와 클라이언트의 sequence number를 알아내는 것이다.

분명히, 가장 흥미로운 것은 클라이언트의 sequence number이다. 왜냐하면, 그것은 클라이언트에 의해 보내진 것처럼 보이는 것을 서버로 데이터를 보내게 할 수 있기 때문이다. 하지만 이 글의 나머지 부분에서는 서버의 sequence number를 먼저 탐지할 필요가 있음을 보여줄 것인데, 이는 클라이언트의 sequence number를 탐지할 필요가 있기 때문이다.

—[3 - 공격 방법 기술(記述)

이 섹션에서는 어떻게 클라이언트의 포트를 확인하는지, 그런 다음 서버의 sequence number를, 그리고 마지막으로 클라이언트의 sequence number를 확인하는 방법에 대해 알아볼 것이다. 우리는 클라이언트의 OS가 취약한 OS라고 가정할 것이다. 그 서버는 어떤 OS에서도 실행될 수 있다.

---

[2] http://insecure.org/nmap/idlescan.html
[3] http://seclists.org/bugtraq/1998/Dec/0079.html

---[3.1 - client-port 찾기

우리는 이미 클라이언트와 서버의 IP, 그리고 서버의 포트를 알고 있다고 가정한다면, 주어진 포트가 정확한 클라이언트의 포트인지 확인하는 잘 알려진 방법이 있다. 이를 위해 우리는 client-IP:guessed-client-port로부터 server-IP:server-port로 SYN 프래그가 설정된 TCP 패킷을 보낼 수 있다.(우리는 이 테크닉이 적용되도록 하기 위해 스푸핑된 IP 패킷을 보낼 수 있을 필요가 있다.)

다음은 guessed-client-port가 정확한 클라이언트의 포트가 아닐 경우 우리가 패킷을 보냈을 때 일어나는 것이다:

```
[=============================================================================]
Attacker (masquerading as client)                          Server

                              --[flags=SYN, SEQ=1000]->

실제 클라이언트

                    <-[flags=SYN+ACK, SEQ=2000, ACK=1001]--

... 실제 클라이언트는 이 연결을 하지 않았으며, 그래서 그것은 RST로 중단한다 ...

                    --[flags=RST]->
[=============================================================================]
```

다음은 guessed-client-port가 정확한 클라이언트의 포트일 경우 우리가 패킷을 보냈을 때 일어나는 것이다:

```
[=============================================================================]
Attacker (masquerading as client)                          Server

                              --[flags=SYN, SEQ=1000]->

실제 클라이언트

... 공격자가 보낸 SYN을 받자마자 서버는 단순한 ACK로 응답한다 ...

                    <-[flags=ACK, SEQ=xxxx, ACK=yyyy]--

... 클라이언트는 단순한 ACK의 응답으로 아무것도 보내지 않는다 ...
[=============================================================================]
```

이제, 이 모든 것 중에서 중요한 것은 첫 번째 경우에서는 클라이언트가 패킷을 하나 보낸다는 것이고, 두 번째 경우에서는 그렇지 않다는 것이다. 만약 섹션 2.2를 주의 깊게 읽었다면 이 특별한 것이 클라이언트의 IP conunter를 확인함으로써 탐지될 수 있다는 것을 독자 여러분들은 알 것이다.

그래서, guessed client-port가 정확한 것인지 확인하기 위해 우리가 해야 할 모든 것은 다음과 같다:

- 클라이언트로 PING을 보낸다.(IP ID를 주목)
- 스푸핑된 SYN 패킷을 보낸다.
- 클라이언트로 PING을 다시 보낸다.(새로운 IP ID를 주목)
- 추측한 포트가 정확한지 확인하기 위해 두 개의 IP ID를 비교한다.

만약 사람들이 효율적인 스캐너를 만들고자 한다면 분명히 많은 어려움들이 있는데, 특히 두 개의 PING 사이에서 클라이언트가 도중에 패킷을 전송할 수 있다는 것과, 클라이언트와 서버 사이의 잠복(latency)(정확하지 않은 추측의 경우 그의 RST 패킷을 클라이언트가 보낸 후에 delay에 영향을 미침)이라는 사실이다. 효율적인 client-port 스캐너를 코딩하는 것은 독자들의 몫으로 남겨둔다. 공격 전에 클라이언트와 서버 사이의 잠복을 측정하고 실시간으로 클라이언트의 트래픽에 자신을 순응시키는 우리의 툴로 클라이언트의 포트를 3분 이내에 보통 발견할 수 있다.

——[3.2 - 서버의 SND.NEXT 발견하기

이제 우리는 클라이언트의 포트를 알게 되었기 때문에 서버의 SND.NEXT(현재 sequence number)를 알 필요가 있다.

어떤 한 호스트가 출발지 및 목적지(source/destination) 포트가 명확하게 있는 TCP 패킷을 받지만 부정확한 seg 그리고/또는 ack를 받을 때마다 정확한 SEQ/ACK number를 가진 간단한 ACK를 보내준다. 우리가 이 문제에 대해 알아보기 전에 RFC793에 정의된 것처럼 정확한 seq/ack[4]이 무엇인지 정의하자:

정확한 SEQ는 패킷을 받는 호스트의 RCV.NEXT와 (RCV.NEXT+RCV.WND-1) 사이에 있는 SEQ이다. 전형적으로 RCV.WND는 아주 큰 번호(최소한 수십 kilobyte)이다.

정확한 ACK는 ACK를 받는 호스트가 이미 보낸 어떤 것의 sequence number에 상응하는 ACK이다. 즉, 어떤 호스트에 의해 받은 패킷의 ACK 필드는 그 호스트 자신의 현재 SND.SEQ보다 낮거나 동일해야 한다. 그렇지 않다면 ACK는 유효하지 않을 것(보내지지 않은 데이터를 인정할 수는 없다!)이다.

---

[4] http://www.ietf.org/rfc/rfc793.txt

sequence number 공간이 "순환적(circular)"이라는 것을 주목하는 것이 중요하다. 예를 들어, ACK의 유효성을 확인하기 위해 받는 호스트에 의해 사용되는 조건은 단순히 unsigned comparison "ACK <= receiver's SND.NEXT"이 아니라 signed comparison "(ACK - receiver's SND.NEXT) <= 0"이다.

자, 우리가 서버의 SND.NEXT를 추측하는 원래의 문제로 돌아가자. 서버에서 클라이언트로 잘못된 SEQ 또는 ACK를 보낸다면 클라이언트는 ACK를 반환할 것이며, 만약 옳다고 추측한다면 클라이언트는 아무것도 보내지 않을 것이라는 것을 우리는 알고 있다. 클라이언트의 포트를 탐지하는 것은 IP ID로 테스트할 수 있다.

만약 |ack1-ack2| = 2^31와 같은 ACK 체크 공식을 살펴보면, 만약 두 개의 ACK 값들(ack1와 ack2이라고 부르자)을 무작위로 선택하면 정확하게 그들 중의 하나는 유효할 것이다. 예를 들어, ack1=0와 ack2=2^31라고 해보자. 만약 실제 ACK가 1과 2^31 사이라면 ack2는 받아들여질 수 있는 ack가 될 것이다. 만약 실제 ACK가 0 또는 (2^32 - 1)와 (2^31 + 1) 사이라면 ack1은 받아들여질 수 있을 것이다.

이것을 고려해보면, 우리는 서버의 SND.NEXT를 찾아내기 위해 sequence number 공간을 더 쉽게 스캐닝할 수 있다. 각각의 추측은 두 개의 패킷을 보내는 것을 포함하고 있는데, 이때 추측된 서버의 SND.NEXT에 설정된 그것의 SEQ 필드와 더불어 패킷이 보내진다. 첫 패킷(resp. second packet)은 ack1(resp. ack2)에 설정된 그것의 ACK 필드를 가질 것이며, 그래서 추측된 것이 SND.NEXT이 정확하다는 것을 확신할 수 있으며, 적어도 그 두 패킷 중에서 하나는 받아들여질 수 있을 것이다. sequence number 공간은 클라이언트 포트 공간보다 더 크지만, 두 개의 사실이 이 스캐닝을 더 쉽게 만든다:

먼저, 클라이언트가 패킷을 받으면, 그것은 즉시 응답한다. client-port 스캐닝에서처럼 클라이언트와 서버 사이에 잠복성을 가진 문제는 없다. 그래서, 두 개의 IP ID 확인 사이의 시간은 아주 작을 수 있으며, 스캐닝 속도를 올리고, 클라이언트가 우리가 확인하는 것과 탐지하는데 바보짓을 하는 것 사이에 IP 트래픽을 가질 가능성을 아주 많이 줄일 것이다.

두 번째, 수신자의 window 때문에 모든 가능한 sequence number를 테스트하는 것이 필요하지는 않다. 사실, 최악의 경우 근사치(2^32 / 클라이언트의 RCV.WND) 추측 계산만 필요하다(이 사실은 이미 [6][5]에서 언급되었다.). 물론 우리는 클라이언트의 RCV.WND를 알지 못한다. 우리는 RCV.WND=64K라고 대략 추측하고, 스캐닝(64K 배수로 SEQ를 시도하며)을 할 수 있다. 그런 다음, 만약 우리가 어떤 것도 발견하지 못했다면 모든 i에 대해 seq = 32K + i*64K와 같은 모든 SEQ들을 시도할 수 있다. 그런 다음 이미 시도한 SEQ들을 다시 테스트하는 것을 피하며 seq=16k + i*32k와

[5] http://osvdb.org/reference/SlippingInTheWindow_v1.0.doc

같은 모든 SEQ 등을 시도하며, window를 좁혀가는 것이다. 전형적인 "modern" 연결에 대해서, 이 스캐닝은 우리 툴로는 15분보다 적게 걸린다.

알려진 서버의 SND.NEXT로, 그리고 ACK를 우리가 모르는 것을 해결하는 방법으로 우리는 "server -> client" 식으로 연결을 하이재킹할 수 있다. 이것은 나쁘지는 않지만 아주 유용한 것은 아니라서, 우리가 클라이언트에서 서버로 데이터를 보내고, 클라이언트가 명령을 실행할 수 있도록 할 수 있는 것 등을 선호할 수 있다. 이를 위해 우리는 클라이언트의 SND.NEXT를 찾아낼 필요가 있다.

——[3.3 - 클라이언트의 SND.NEXT 찾기

클라이언트의 SND.NEXT를 알아내기 위해 무엇을 할 수 있는가? 분명 우리는 서버의 SND.NEXT에 했던 것과 같은 방법을 사용할 수 없는데, 이는 서버의 OS가 아마도 이 공격에 취약하지 않기 때문이며, 게다가 서버 상의 부담스러운 네트워크 트래픽은 IP ID 분석을 실행 불가능하게 할 수 있다.

하지만, 우리는 서버의 SND.NEXT를 알고 있다. 또한 클라이언트의 SND.NEXT가 클라이언트의 내부로 들어오는 패킷의 ACK 필드를 점검하는데 사용된다는 것을 알고 있다. 그래서 우리는 서버의 SND.NEXT에 설정된 SEQ 필드로 서버로부터 클라이언트로 패킷을 보낼 수 있으며, ACK를 가려내고, 그리고 (다시 IP ID로) ACK가 받아들일 수 있는지 확인할 수 있다.

만약 ACK가 받아들일 수 있는지 탐지한다면 이것은 (guessed_ACK - SND.NEXT) <= 0라는 것을 의미한다. 그렇지 않다면 이미 추측했을 것 같은데, (guessed_ACK - SND_NEXT) > 0임을 의미한다.

이 지식을 이용하면, 우리는 바이너리 분석을 통해 약 32번의 시도로 정확한 SND_NEXT를 알아낼 수 있다(sequence space가 순환적이기 때문에 다소 변경된 것이다).

이제 우리는 마침내 필요한 모든 정보를 가지게 되었으며, 우리는 클라이언트나 서버로 어느 쪽으로부터도 세션 하이재킹(session hijacking)을 할 수 있게 되었다.

—[4 - 토론

이 섹션에서 우리는 취약한 시스템을 확인하고, 이 공격의 한계를 토론하며, 더 오래된 시스템에 대한 비슷한 공격을 제시할 것이다.

----[4.1 - 취약한 시스템

이 공격은 Windows 2K, Windows XP <= SP2, 그리고 FreeBSD 4에서 테스트되었다. FreeBSD는 IP ID를 랜덤화하는 커널 옵션을 가지고 있으며, 이것은 이 공격을 불가능하게 만든다는 것을 주목해야 한다. 우리가 알고 있는 한 Windows 2K와 XP에 대한 해결책은 없다.

취약한 시스템에서 이 공격을 가능하게 만드는 유일한 "버그"는 비랜덤화된 IP ID이다. 다른 행위자(우리가 바이너리 분석 등을 가능하게 하는 ACK 점검)들은 RFC793에 의해 예상된다(하지만, [4][6]에서 이 문제를 개선시키기 위한 작업이 있었다).

우리가 테스트를 할 수 있었던 한 Windows 2K, Windows XP, 그리고 FreeBSD 4가 취약했다는 것을 확인하는 것은 흥미롭다. 같은 IP ID 증가 시스템을 사용하는 다른 OS가 있지만 같은 ACk 점검 메커니즘을 사용하지는 않는다. Windows와 FreeBSD의 TCP/IP 스택 행위 사이의 유사성은 골칫거리다. MacOS X는 FreeBSD에 기반을 두고 있지만 취약하지는 않은데, 이것은 MacOS X가 다른 IP ID 번호부여(numbering) 구조를 사용하기 때문이다. Windows Vista는 테스트를 하지 않았다.

----[4.2 - 한계들

앞에서 기술한 공격은 다양한 한계들을 가지고 있다:

첫째, 이 공격은 Windows 98에서 작동되지 않는다. 하지만 실제 한계는 아닌데, 왜냐하면 Windows 98의 초기 SEQ는 1,000분의 1초, 2^32 이하(modulo 2^32) 내의 머신의 가동시간과 동일하기 때문이다. 우리는 Windows 98로 어떻게 하이재킹을 하는지에 대해 토론하지 않을 것이다.
두번째, 만약 클라이언트의 연결이 느리거나 많은 트래픽을 가지면 (IP ID 분석으로 바보 같은 짓을 하며) 이 공격은 어려워질 것이다. 또한, 클라이언트와 서버 사이의 잠복성의 문제가 있다. 이 문제들은 잠복을 측정하고, 호스트들이 트래픽을 가지는 때를 탐지하는 똑똑한 툴을 만들어 줄일 수 있다.

더 나아가, 우리는 클라이언트 호스트에 접근할 필요가 있다. 우리는 패킷을 보내고 IP ID를 가지기 위해 응답을 받을 수 있을 필요가 있다. ICMP 또는 TCP 또는 그 무엇이든 어떤 타입의 패킷도 괜찮다. 이 공격은 만약 호스트가 모든 타입의 패킷을 절대적으로 막는 방화벽/NAT/... 등의 뒤에 있을 경우 가능하지 않을 것이다. 하지만 필터링 되지 않는

[6] http://www.ietf.org/internet-drafts/draft-ietf-tcpm-tcpsecure-07.txt

하나의 포트만 있어도 이 공격을 가능하게 만들 수 있기에 충분하다. 이 문제는 통합된 방화벽을 가진 Windows XP SP2와 그 이후에 나온 OS에도 존재한다. Windows XP SP2는 취약하지만 방화벽이 어떤 상황에서는 공격을 막을 수 있으지 모른다.

—[5 - 결론

이 글에서 우리는 Windows 2K/XP, 그리고 FreeBSD 4에서 작동하는 blind TCP hijacking의 방법 한가지를 제시했다. 이 방법이 많은 한계를 가지고 있지만 완벽하게 가능하며, 많은 호스트들에게 적용될 수 있다. 더 나아가, TCP 상의 많은 프로토콜들은 여전히 암호화되지 않은 통신을 사용하며, 그래서 blind TCP hijacking의 보안에 대한 영향을 무시할 수는 없다.

—[6 - 참고문헌

[1] http://lcamtuf.coredump.cx/newtcp/

[2] http://www.ietf.org/rfc/rfc793.txt

[3] http://insecure.org/nmap/idlescan.html

[4] http://www.ietf.org/internet-drafts/draft-ietf-tcpm-tcpsecure-07.txt

[5] http://seclists.org/bugtraq/1998/Dec/0079.html

[6] http://osvdb.org/reference/SlippingInTheWindow_v1.0.doc