BPF JIT spray review

Reviewer	Jinho Ju(hackin / wnwlsgh98@gmail.com)
Date created	2022.11.12

INDEX

1. Introduction
2. Background
3. JIT spray attack
    3.1. Original JIT spray attack on LInux
        3.1.1. Creating the BPF payload
        3.1.2. Loading the payload in memory
        3.1.3. Redirection of execution
        3.1.4. The attack
    3.2 Community response
        3.2.1. Upstream Linux kernel fix
        3.2.2. Grsecurity fix
4. Our attack
    4.1. Approach 1
    4.2 Approach 2
5. Mitigation measures
6. Reference

1 Introduce

오늘날 점점 더 많은 공격자들은 userspace application 중 mobile & embedded devices보다 kernel을 공격하는 것에 대해 집중하고 있다. 이에 대한 주된 이유는 userspace application이 exploit될 때 발생하는 피해를 줄이기 위해 몇 년에 걸쳐 수행된 광범위한 작업 때문이다. 예를 들어, Android의 최신 배포들에는 손상된 application이 OS 자체에 대해 중요한 제어를 할 수 없도록 잘 설계된 SEAndroid의 정책들이 반영되어 있다. 이와 반대로, kernel에서 vulnerability를 찾으면 거의 항상 장치의 모든 부분이 손상된다.

많은 kernel(및 userspace) vulnerability들은 uninitialized variables, missing boundary checks, use-after-free 등과 같은 프로그래밍 실수로 인한 결과이다. 이러한 실수를 찾는 것에 도움을 주는 도구를 개발하는 것은 중요하지만, 이는 완전히 실수들로 일해 발생하는 결과들을 피할 수 없다. 더 나아가, upstream kernel에서 vulnerability가 발견되어 수정되더라도 모든 user devices에 수정 사항들이 적용되기까지는 대략 5년 정도 걸린다.

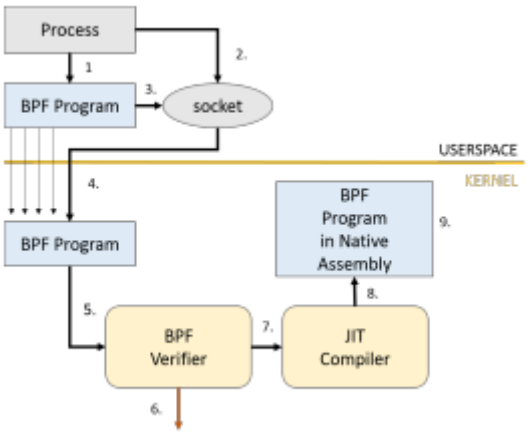
Kernel Self Protection Project(이하 KSPP)는 kernel 내부에 다양한 강화 메커니즘을 구현하여 성공적인 exploit으로 이어질 수 있는 모든 종류의 vulnerability를 제거하기 위한 시도들을 했다. 특정 추가 보호 메커니즘의 필요성을 입증하는 Proof of Concept(이하 PoC)공격을 생성하는 것이 해당 프로젝트의 중요한 부분이다. 이는 kernel 하위 시스템의 관리자로부터 폭 넓은 허가를 얻는 데 도움이 되기 때문이다.

Linux kernel Berkeley Packet Filter(이하 BPF) Just-In-Time(이하 JIT) compiler는 kernel에서 널리 사용되고 과거에 성공적인 공격에 사용되었기에 프로젝트의 중요한 부분이었다. Linux BPF/JIT에 대한 JIT spray 공격은 2012에 제시되었다. 또한, 일부 대응책은 Linux kernel v3.10에서 구현되었다. 해당 문서에서는 Linux kernel v4.4에서 대응책들을 우회할 수 있는 방법에 대해 서술하였다. 또한 이러한 유형의 공격들을 시행할 수 없도록 Linux kernel에 추가된 최근의 조치들에 대해서도 설명한다.

2 Background

Networking을 지원하는 초기 버전의 UNIX에서는 빠른 nerwork packet검사와 모니터링이 필요했다. 이를 위해 kernel packet filter agent라는 개념을 도입하여 속도를 높이고 kernel과 userspace 사이의 불필요한 packet 정보에 대한 복사를 피했다. 다른 UNIX 기반 OS들은 이러한 agent의 자체적인 버전을 구현하였다. 추후에 Linux에서 채택한 솔루션은 BPF라고 불리는 1993년에 도입된 BSD packet filter이다. 이 agent를 사용한다면 userspace program이 filter program을 socket에 연결하고 socket을 통해 들어오는 특정 데이터 흐름을 빠르고 효과적인 방식으로 제한할 수 있다.

Linux BPF는 본래 filter를 프로그래밍하는데 사용할 수 있는 instruction set을 제공하였다. 이는 현재의 classic BPF(cBPF)이다. 추후에 더욱 다양한 instruction set이 도입되어 extended BPF(eBPF)라고 불린다. 해당 문서에서 용어를 단순화하기 위해 이후에 나오는 instruction set을 단순히 BPF instruction이라 표현할 것이다. Linux BPF는 몇 개의 레지스터, 스택, implicit program counter로 구성된 최소한의 virtual machine으로 볼 수 있다. 유효한 BPF program내에서는 packet에서 데이터 가져오기, 상수 및 입력 데이터를 활용하는 산술 연산, 상수 또는 packet data에 대한 결과 비교와 같이 다양한 작업들을 수행하도록 허용된다. Linux BPF 하위 시스템에는 BPF program의 정확성으 확인하는데 사용되는 verifier라고 하는 특수 구성 요소가 있다. 모든 BPF program은 실행되기 전에 이 구성 요소의 승인을 반드시 받아야 한다. verifier은 BPF program의 모든 분기를 탐색하고 분석하는 정적 코드 분석기이다. 이는 unreachable instruction, out of bound jump, loop 등을 감지하려고 시도한다. 또한, verifier은 BPF program의 최대길이를 4096개의 BPF instruction으로 제한한다.



본래의 목적은 network packet filtering 용으로 설계되었지만 현재 Linux BPF는 seccomp, tracing, Kernel Connection Multiplexer(KCM)의 system call filtering을 포함하여 다른 많은 영역에서 활용된다. Linux는 packet filtering 성능을 향상시키기 위해 JIT compiler를 사용하여 BPF instruction을 기본 시스템 어셈블리어로 변환한다. JIT compiler는 x86 및 ARM을 포함한 모든 주요 architecture에 제공된다. 이 JIT compiler는 Ubuntu 또는 fedora와 같은 표준 Linux 배포판에서는 기본적으로 활성화되지 않지만 일반적으로 라우터와 같은 네트워크 장비에서는 활성화된다.

Figure1은 BPF program이 Linux kernel에서 적재되고 처리되는 방식은 간략하게 보여준다. 먼저 userspave 프로세스는 socket과 BPF program을 만들고, program을 socket에 연결한다(1~3). 다음으로, program은 kernel로 전송되어 적재되기 이전에 안전한지 확인하기 위해 verifier에 들어간다(4~5).

3 JIT spray attack

JIT spraying란 공격자가 만든 payload를 OS의 실행가능한 메모리 영역에 JIT을 이용하여 적재하기 위한 공격기법이다. 이 기법은 일반적으로 constant로 incoding된 payload명령을 JIT compiler로 전달한 후, 적절한 OS bug를 활용하여 payload code를 가리키게 함으로써 공격의 목적을 이룬다.

일반적으로 payload는 정확히 알 수 없거나 공격자에 의해 위치가 정해지므로, payload를 OS memory에 최대한 많이 복사하여 spray함으로써 성공할 확률을 극대화한다. 해당 공격기법이 위험한 이유는 JIT compiler때문이다. JIT compiler는 자체적인 특성으로 인해 일반적으로 NX(Non-Executed) bit 와 같은 다양한 데이터 실행 방지 기술에서 제외되기 때문이다. 또한, 특히나 x86 architecture에서 성공률이 높은데, 이는 unaligned instruction execution을 지원하기 때문이다. 여기서 unaligned instruction execution란, multi-byte machine instruction의 중간으로 점프하고, 그 점프한 지점부터 실행할 수 있는 것을 말한다. x86 architecture는 instruction의 길이가 1byte~15byte사이일 수 있고, 프로세서는 순서에 관계없이 정확하게 실행될 수 있어야 하기에 unaligned instruction execution기능을 지원한다.

해당 공격기법은 2010년에 Dion Blazakis에 의해 처음 소개되었고, Windows에서 Adobe Flash player에 대한 공격에 처음 활용되었다.

3.1 Original JIT spray attack on LInux

BPF JIT compiler를 사용하는 LInux kernel에 대한 최초의 JIT spray attack에 대한 PoC 2012년 Keegan McAllister에 의해 소개되었다. 이를 활용한 PoC exploit code는 Linux device에서 root shell을 획득할 수 있다.

3.1.1 Creating the BPF payload

PoC는 "commit_creds(prepare_kernel_cred(0))"를 포함하는 payload는 유효한 BPF progam을 생성한다. 이는 Linux에서 root권한을 얻는 일반적인 exploit방법이다.
"commit_creds(prepare_kernel_cred(0))"함수를 이용하여 구성한 코드를 통해 현재 프로세스가 가지는 권한을 root로 설정한다는 의미이다. commit_creds의 주소와 "prepare_kernel_cred" kernel symbol들은 "/proc/kallsyms"라는 kernel interface를 통해 확인할 수 있다. payload 명령어들은 BPF load immediate instruction(BPF_LD+BPF_IMM)을 사용하는 filter program에 내장된다.

BPF load immediate instruction은 4byte 상수로 standard register(x86기준 eax)에 적재된다. 해당 instruction이 compile 될 때, x86기준으로 `mov $x, %eax instruction`으로 변환된다. 이는 다음과 같은 바이트 순서를 따른다.

```
b8 xx xx xx xx
```

b8은 opcode instruction이며 다음의 4byte는 명령어 인자인 `$x`이다. 공격자가 이 4bytes를 자유롭게 세팅할 수 있다고 할 때, 실제로 사 할 수 있는 것은 오직 첫 3개만이 임의로 선택될 수 있다. 마지막 byte는 이후에 오는 b8 instruction opcode와 결합되는데, 이 시점은 정렬되어 있지 않은 실행이며 무해한 instruction이 생산된다. 이 목적을 위해 마지막 byte는 a8로 결정된다. a8 b8의 바이트 순서는 무해한

`test $0xb8, %al x86` instruction을 나타낸다. BPF load immediate instruction이 여러번 반복될때의 byte sequence는 다음과 같다.

```
b8 xx xx xx a8 b8 xx xx xx a8 b8 ...
```

figure2는 어떻게 payload가 BPF 의사코드가 x86기계어로 JIT compiler에서 변환되어 사용되는지, payload의 두번째 byte부터 시작될 때 어떻게 기계어의 정렬되지 않은 실행이 보여지는지를 나타낸다.

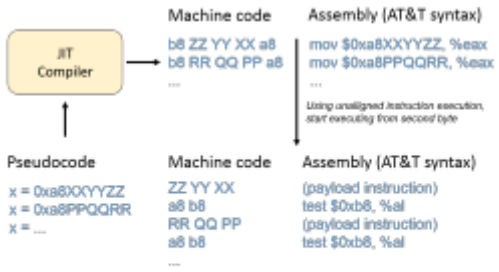


Figure 2: BPF payload JIT compilation and un-aligned execution

cf. test: operand 두개에 대해 and연산, 플래그값을 세팅하여 분기문(ex. JE, JZ,...)에 영향을 준다. 중요한 명령은`test EAX EAX`와 같은 명령어이다. 같은 레지스터끼리 연산을 하는 경우 어차피 EAX값이 나오겠지만, 이는 사실 EAX의 값이 0인지 아닌지를 판단하기 위해서이고, EAX가 0인 경우, 결과는 0이고 Z 플래그(ZF가 세팅)

3.1.2 Loading the payload in memory

Kernel memory에 BPF filter payload의 많은 복사본을 load하기 위해서는 공격자의 process가 많은 local socket을 생성해야한다. 각 소켓에는 하나의 BPF filter만 연결할 수 있기 때문이다. Linux가 주어진 시간에 프로세스가 가질 수 있는 열린 file discriptor(fd)의 갯수를 제한할 때, McAllister는 이 제한을 bypass하기 위해 특별한 트릭을 활용하였다. 이 트릭은 하나의 local Unix socket을 열고, resulting file discriptor를 다른 local Unix socket으로 보낸 다음 원래 socket을 닫는 것이다. 따라서 소켓은 process socket 생성 제한에 포함되지 않지만 이와 상관없이 kernel memory에 유지된다. 이 방법을 이용하여 McAllister은 8000개의 소켓을 만들고 kernel memory payload를 포함하는 8000개의 BPF filter를 load하였다.

3.1.3 Redirection of execution

마지막으로 PoC code는 proc 가상파일시스템에서 제공하는 인터페이스를 사용하여 userspace process에 의해 지정된 주소로 점프하는 kernel module(jump.ko)를 포함하고 있다. 이 모듈은 실행 흐름을 redirect하기 위해 실제로 kernel에서 bug를 찾는 작업을 simulation 한다. jump.ko는 공격하기 전에 root권한을 이용하여 load해야 하며, 이는 root권한을 얻기 위한 공격자는 사용 불가하다. 이 kernel module은 단순히 JIT spray 공격이 작동하는 것을 보여주기 위한 entry point를 제공하기 위해 사용된다.

3.1.4 The attack

공격자의 프로그램은 payload를 포함하는 8000개의 filter로 kernel memory를 채운 후 kernel module mapping memory 공간내의 임의의 page로 점프하고 미리 정의된 offset에서 payload를 실행하려고 시도하는 loop를 시작한다. 공격 성공의 핵심은 v3.10보다 오래된 kernel에서 BPF JIT compiler가 memory page시작부분에 각 filter에 할당했다는 사실이다. BPF filter의 길이가 고정되어 있기 때문에 공격자가 payload에 도달하기 위해 page에서 점프할 정확한 offset을 항상 알고 있다. 각 추측에 대한 시도는 자식 프로세스에 의해 수행된다. 이렇게하면 잘못된 page를 방문하고 잘못된 instruction을 실행할 가능성이 있는 경우 Linux kernel에 의해 자식 프로세스만 종료되고 부모 프로세스는 계속 공격할 수 있다.

공격자가 filter가 포함되지 않는 page로 jump할 때, 시스템의 동작을 예측할 수 없다는 점에 유의하는 것이 중요하다. 대부분의 경우 landing instruction이 유효하지 않거나 무해한 경우 프로세스가 단순히 종료되고 공격이 계속될 수 있다. 그러나 instruction이 주요 시스템의 일부 register를 변조하면 전체 OS가 중단될 수 있으며 시스템을 복구하기 위해서는 hard reboot가 필요하다.

3.2 Community response

공격이 공개된 이후 Linux kernel community는 다양한 대응책을 마련하였다.

3.2.1 Upstream Linux kernel fix

upstream linux kernel은 page 내에서 BPF program이 load되는 주소를 무작위로 지정하는 patch set을 병합하였다. filter는 page의 시작부분에서 시작하는 대신에 page 내부의 임의의 offset에 위치한다. 또한, page의 시작과 filter 사이의 공간(이하 hole)은 공격자가 실행할 경우 시스템을 중단시키는 것을 목적으로 하는 architecture별 instruction으로 채워져 있다. x86의 경우 hole이 반복되는 INT3(0xcc) instruction으로 채워져 Linux kernel에서 SIGTRAP interrupt가 발생한다. 이 접근 방식은 exploit의 성공 확률을 현저하게 낮췄다. 이제 공격자는 정확한 page를 추측할 수 없을 뿐더러, filter가 시작되는 4KB 크기의 page 내부의 정확한 offset도 추측해야 하기 때문이다.

더 나아가, 공격자가 filter 복사본이 포함된 page로 이동할 때 잘못된 offset을 추측하면, 치명적인 대가를 치루게 된다. INT3 instruction을 실행하면 허용되지 않는 instruction을 실행하는것보다 더 심각한 결과로 이어진다. 이 심각한 결과는 일반적으로 kernel panic과 OS freeze를 뜻한다. 현재로써 이 결과에 대한 원인은 파악되지 않고 있으며, 이 경우 kernel동작에 대한 조사가 계속 이루어지고 있다.

3.2.2 Grsecurity fix

Grsecurity3로 알려진 또 다른 kernel security project는 exploit을 방어하기 위해 다른 강화 매커니즘을 발표했다. "constant blinding"이라 칭하는 기술을 사용하여 공격자가 payload instruction을 상수로 load하는 것을 방지하고, 소스코드에 code injection을 수행할 수 있는 벡터를 차단하였다. "constant blinding"은 상수를 memory에 있는 그대로 저장하지 않고, 생성된 난수와 XOR 후 저장하는 아이디어이다. 합당한 작업을 통해 상수에 접근해야하는 경우 정확한 값을 얻기 위해서는 난수와 다시 XOR할 수 있다. 이 기능은 x86 architecture에서만 지원하도록 구현되었다. 여러 이유들로 인하여 upstream kernel로는 병합되지 않았다. 그 이유에는 architecture의 독립적인 접근 방식, 기능의 성능적 영향, 다양한 정치적 이유가 있으며 무엇보다 upstream kernel에서 구현된 randomization 조치만으로도 BPF JIT spray 공격을 차단하기 충분할 것이라 생각하였기 때문이다. 실제로, 현재까지 강화된 upstream kernel에 대한 공격은 공식적으로 입증되지 않았다.

4 Our attack

KSPP의 일환으로 Kernel BPF/JIT 유지 관리자 중 한 명인 Daniel Borkmann과 저자들은 BPF/JIT보안을 더욱 강화하기 시작하였으며, Grsecurity에서 제안한 "constant blinding" 접근 방식에 대해서도 지속적으로 고려했다. 저자들의 목표는 upstream kernel에 구현된 기존의 조치가 JIT spray 공격을 차단하기에 불충분하다는 것을 증명하는 것이었다. Upstream kernel의 BPF/JIT에 대한 공격을 보여줄 수 있다면, upstream kernel에 추가적인 보호기능들이 병합될 가능성이 높아지기 때문이다. 저자들의 목표는 버전의 공격을 개발하는 것이었다.

작업의 주요한 부분은 2015년 말~ 2016년 초에 ubuntu 15.10에서 당시 사용가능한 최신 안정 커널(v4.4.0-rc5)이 기본 Ubuntu구성으로 compile되어 KVM(Kernel-based Virtual Machine)에서 구동되는 방식으로 실행되었다. 전체 설정은 x86_64 architecture에 대해 수행되었습니다. 저자들은 두가지 다른 공격 접근 방식을 개발했다. 저자들이 우선적으로 해결해야할 문제는 /proc/kallsyms kernel interface에서 사용하는 kernel symbol들(ex, commit_creds, prepare_kernel_creds)의 위치를 얻을 수 없다는 것이었다. v4.4에서 이미 kernel pointer protection이 구현되어 있기 때문이고, 이는 userspace applications에 대한 kernel pointer의 값을 숨긴다. 이 protection은 kptr_restrict 옵션을 0으로 세팅함으로써 해제할 수 있지만, root권한이어야만 가능하다. 그렇기에 이를 해결할 수 있는 방법은 kptr_restrict가 비활성화된 시스템에서 이러한 symbol들의 주소에서 얻은 이후에 특정 kerenl 버전에 대해 symbol들의 주소를 hardcoding하는 것이다. 이것은 현재 Ubuntu 및 kernel v4.4가 있는 유사한 배포판에서는 KASLR을 사용하지 않기 때문에 가능하다. 따라서 kernel symbol들은 compile된 특정 kernel의 모든 복사본들에 대한 결정적인 주소에 있다. 이후에 runtime동안 저자들의 공격은 table에서 machine kernel version을 조회하여 정확한 symbol 주소들을 추출해낼 수 있다.

4.1 Approach 1

BPF filter program의 크기는 4096개의 BPF instructions로 제한되어 있지만, 이것은 kernel memory page의 4KB보다 크고, compile된 BPF filter를 얻기엔 충분하다. 만약 filter의 크기가 한개의 page보다는 크지만 2개의 page보단 작은 크기를 가지게 되고, filter program이 포함된 page의 시작부분으로 점프하는 경우의 50%는 program instruction에 도달한다는 것을 확신할 수 있다. BPF명령어의 수를 최대값 4096으로 늘려 filter를 2개의 page보다 길게 확장하면 확률이 훨씬 더 높아질 수 있다. 그러나 실제로는 2560 BPF instruction보다 큰 filter는 setsocket()에 의해 "Out of memory error"를 나타내며 거부된다.

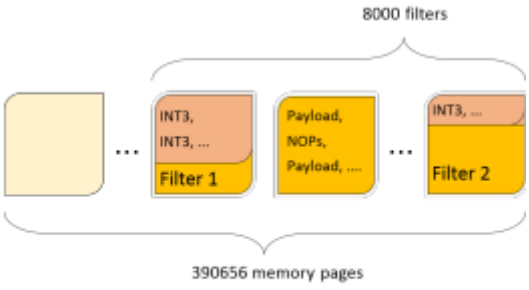


Figure 3: Attack approach 1

Approach 1에서, 저자들은 기존의 공격을 변경하여 2471개의 BPF instruction(program의 길이: 12439)을 포함하는 더 큰 BPF program을 생성하고, 3~4개의 4KB의 페이지를 차지하게 된다. payload의 시작과 끝을 16byte(BPF header 기준) 로 정렬하기 위해 padding으로 사용되는 추가 NOP instruction으로 payload를 여러번 반복하여 이를 수행했다. Figure 3은 이러한 공격 접근에 대해 보여준다.

임의의 page로 이동할 때, 다음 임의의 page로 이동하기 이전에 첫 10개의 offset을 실행하려고 한다. 10개의 offset은 각 payload 사이에 padding으로 추가되는 최대 NOP instruction 수에 해당한다. 이제 소켓의 수와 적재된 BPF filter의 개수는 원래의 공격과 동일하게 8000개가 된다. filter의 복사본을 포함하는 page로 점프하지만 위치가 page의 시작이 아닐때, 저자들은 INT3 instruction으로 채워진 hole에 의해 VM 중단이 발생하고, 공격에 실패한다.

4.2 Approach 2

다음 접근 방식은 BPF filter program의 할당이 어떻게 발생하고, BPF filter의 임의의 offset이 어떻게 계산되는지를 기반으로 한다. 이는 Listing 1에서 볼 수 있는 bpf_jit_binary_alloc()에 의해 수행된다. 이 함수는 먼저 program에 할당될 모든 메모리의 크리를 계산한다(line 223). 여기서 proglen은 실제 BPF program의 길이(byte단위), sizeof(*hdr)은 4byte, PAGE_SIZE는 4096 byte이다. 이 모든 공간들은 illegal architecture dependant instructions(x86의 경우 INT3)에 의해 미리 채워진다(line 229). BPF filter가 실제로 시작되는 offset은 마지막에 계산된다(line 234). 방금의 단계에서 추론할 수 있는 것은 proglen을 PAGE_SIZE -128 -sizeof(hdr)로 만들 수 있다면, BPF filter에 할당된 한 개의 page만 남게되며 page의 시작 부분에 생기는 hole의 최대 크기는 128이 된다. Hole의 실제 크기는 무작위이기에 알 수 없지만, 최대 크기는 128로 정적이다. offset 132(128 + sizeof(*hdr))에서 점프하면 payload에 도착하는 것을 보장한다. 이러한 방법으로 삽입된 INT3 instruction과 부정적인 영향들을 완전히 피할 수 있다. Figure 4는이러한 공격의 접근 방식을 보여준다.

실험에서 저자들은, filter의 크기를 3964 bytes로 가져오고, hole_offset이라 불리는 첫 132 bytes를 성공적으로 건너뛸 수 있었다. Hole_offset과 hole_offset+1을 모두 시도하면, 선택된 점프 대상에 BPF load immediate instruction으로부터 파생된 b8 byte가 포함되는 불행한 경우로부터 보호할 수 있다. b8로 점프하는 것은 payload가 아니라 실제로는 `mov %eax, xxxxxxx` 이라는 instruction을 실행하는 것을 의미한다. 인접한 두 offset으로 점프하면 적어도 둘 중 하나는 b8을 포함하지 않을 것이라 보장할 수 있다.

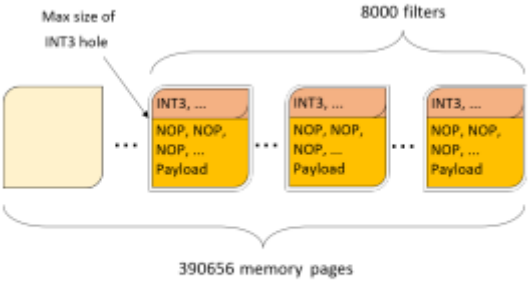


Figure 4: Attack approach 2

Listing 1: bpf_jit_binary_alloc() function from kernel/bpf/core.c

```
struct bpf_binary_header *
bpf_jit_binary_alloc(unsigned int proglen, u8 **image_ptr,
                     unsigned int alignment,
                     bpf_jit_fill_hole_t bpf_fill_ill_insns)
{
        struct bpf_binary_header *hdr;
        unsigned int size, hole, start;

        /* Most of BPF filters are really small, but if some of them
         * fill a page, allow at least 128 extra bytes to insert a
         * random section of illegal instructions.
         */
        size = round_up(proglen + sizeof(*hdr) + 128, PAGE_SIZE);
        hdr = module_alloc(size);
        if (hdr == NULL)
                return NULL;

        /* Fill space with illegal/arch-dep instructions. */
        bpf_fill_ill_insns(hdr, size);

        hdr->pages = size / PAGE_SIZE;
        hole = min_t(unsigned int, size - (proglen + sizeof(*hdr)),
                     PAGE_SIZE - sizeof(*hdr));
        start = (get_random_int() % hole) & ~(alignment - 1);

        /* Leave a random number of instructions before BPF code. */
        *image_ptr = &hdr->image[start];

        return hdr;
}
```

5 Mitigation measures

공격이 개발된 이후로 많은 보호기법들이 upstream kernel에 병합되었다. Kernel Self Protection Project의 일부로써 공격과 병행하여 Daniel Borkmann이 개발한 주요 조치는 eBPF에서 상수들을 블라인드할 수 있도록 upstream지원을 하는 것이다. x86 architecture에만 적용될 수 있는 GRsecurity의 구현과 달리 Daniel의 조치는 거의 모든 architecture에 독립적인 구현을 제공한다. 이는 이미 eBPF instruction수준에 있는 상수를 블라인드하고 JIT compiler에 블라인드된 상수를 제공하여 얻는다. 이를 통해 모든 architecture에서 BPF/JIT의 강화를 위한 통합되고 견고한 설계를 가질 수 있을 뿐 아니라, 잘 검토된 강화 구현을 통해 보안성을 더욱 향상 시킬 수 있었다. 이 보호 기법은 2016년 5월에 upstream kernel에 병합되었으며 v4.7의 일부로 배포되었다.

Unix domain socket을 악용하는 것을 방지하기 위해 더 강화되었습니다. Willy Tarreay는 열려있는 fd에 대한 자원 제한을 우회하는 트릭에 대해 차단하는 patch를 병합했다. 이 보호 기법은 v4.5의 일부로 kernel에 배포되었다.

Kernel symbol의 정적 위치에 의존하는 exploit을 방지하기 위한 주요한 기능인 x86_64용 Kernel Address Space Layout Randomization(KASLR)은 v4.8의 일부로 배포되었다. 활성화되어 있는 경우, 이 기능은 압축해제된 kernel image의 physical memory와 virtual memory의 위치를 무작위로 지정하고, 공격자가 공격에 필요한 kernel symbol의 위치를 발견하는 것을 매우 어렵게 만든다. 예를 들어, 더 이상 commit_creds()의 특정 binary 위치에 의존하거나 kernel 버전을 기반으로 kernel_read() symbol을 준비하는 것이 불가능하다. 공격자는 이러한 값을 얻기 위해 다양한 information leak을 사용해야 한다.

KASLR은 중요하지만, 여전히 모든 exploit으로부터 완전히 보호할 수는 없다. 예를 들어, vmalloc()이 kernel memory를 할당하는 주소는 여전히 무작위로 정해지지 않기에 공격을 원활하게 하기 위한 추가적인 정보를 제공할 수 있다.

6 Reference

https://www.blackhat.com/docs/eu-16/materials/eu-16-Reshetova-Randomization-Can't-Stop-BPF-JIT-Spray-wp.pdf