CPython Copy-and-Patch JIT Compiler 메커니즘의 보안 취약점 분석

조민혁[01], 정민서, 강수빈, 임정민, 유시환
단국대학교 모바일시스템공학과
{cgumgek8, jminseo1832, kang091304, jun9min1782}@dankook.ac.kr

Security Vulnerability Analysis and Exploitation of the Copy-and-Patch JIT Compiler Mechanism in CPython

Minhyuk Cho[01], Minseo Jeong, Subin Kang, Jungmin Yim, Seehwan Yoo
Dept. of Mobile Systems Engineering, Dankook University

요 약

본 논문은 Python의 Copy-and-Patch JIT 컴파일 환경에서 발생할 수 있는 보안 취약점을 분석하고, 이를 악용한 공격 시나리오를 설계·검증하였다. ARM 아키텍처 기반 리눅스 환경에서 Python Internal API 및 시스템 콜을 이용해 JIT 메모리 주소를 유출하고 실행 권한을 조작함으로써 쉘코드 실행에 성공하였으며 Copy-and-Patch JIT 구조가 실질적 보안 위협이 될 수 있음을 실험적으로 입증하였다. 향후 x86 환경에 대한 확장 실험을 통해 아키텍처별 대응 기법을 제안할 예정이다.*

1. 서 론

Python은 높은 생산성과 광범위한 생태계를 바탕으로 다양한 산업 분야에서 널리 사용되고 있으며 특히 머신러닝, 웹 개발, 보안 자동화와 같은 고성능 연산을 요구하는 영역에서도 핵심 도구로 활용되고 있다. 그러나 Python은 전통적으로 인터프리터 기반의 실행 방식을 사용하기 때문에 반복 연산 시 성능 병목이 발생하며 이를 해결하기 위한 다양한 Just-In-Time(JIT) 컴파일 기법이 연구되고 있다[1]. 특히 최근 Python 커뮤니티는 Python 3.13부터 공식적으로 JIT 컴파일을 지원하기 시작했다[2]. 이는 Copy-and-Patch JIT으로 지원되는데 기존 JIT 컴파일 방식이 바이트코드를 기계어로 변환하는 방식이었다면 Copy-and-Patch JIT은 미리 만들어진 기계어 조각을 복사하여 패치하는 방식으로 성능을 약 10% 향상시킨 것에서 의미가 있다[3]. 하지만 이와 같은 코드 생성 방식은 보안적인 측면에서 새로운 공격 표면을 제공할 수 있다. 특히 Copy-and-Patch JIT은 코드 블록을 메모리에 복사한 후 실행 가능하도록 패치하는 과정에서 시스템 콜을 사용하여 실행 권한을 동적으로 설정한다. 이 과정은 실행 흐름 제어 및 메모리 권한 변경이라는 두 가지 측면에서 악의적인 코드 삽입과 실행이 가능한 잠재적 취약점으로 작용할 수 있다. 본 논문에서는 CPython 런타임 환경에서 동작하는 Copy-and-Patch JIT의 보안 취약점을 분석하고, 해당 구조가 공격 흐름의 진입점으로 악용될 수 있음을 실험적으로 입증한다. 구체적으로 사용자 권한 수준의 공격자가 Python 내부 API를 통해 JIT 코드 메모리를 식별하고 mprotect 시스템 콜을 사용하여 RWX 권한을 부여한 뒤, trampoline 조작 및 JIT Spraying을 통해 쉘을 획득하는 과정을 ARM 기반 리눅스 시스템에서 성공적으로 구현하였다. 이를 통해 Copy-and-Patch JIT이 제공하는 성능 최적화가 보안성 저하로 이어질 수 있는 구조적 위험 요소임을 밝히고, Python 기반 시스템 전반에 대한 보안적 재고가 필요함을 제안한다.

2. 배경지식 및 관련 연구

2.1 CPython

CPython은 Python 언어의 공식 구현체로 C 언어로 작성되어 있으며 가장 널리 사용되는 인터프리터다. Python 코드는 실행 시 먼저 바이트 코드로 변환된 후 CPython 내부의 평가 루프에서 해석되며 실행된다. 이 구조는 단순성과 이식성 측면에서 강점을 가지지만 반복되는 연산이 많은 코드에서는 성능 저하가 발생할 수 있어 이를 보완하기 위한 다양한 JIT 컴파일 기술이 제안되었다. CPython은 기본적으로 JIT을 포함하지 않지만 Copy-and-Patch JIT과 같은 외부 확장 방식이 연구되고 있으며 이는 반복 실행되는 바이트코드를 네이티브 코드로 치환하고 해당 메모리를 실행 가능하도록 설정하여 성능을 개선한다.

2.2 Copy-and-Patch JIT

Copy-and-Patch JIT은 CPython에서 시도된 실험적 JIT 컴파일 방식으로 바이트코드 인터프리팅의 성능 병목

* 본 연구는 2024년 과학기술정통신부 및 정보통신기획평가원의 SW중심대학사업 지원을 받아 수행되었음(2024-0-00035)

을 해소하기 위해 설계되었다. 이 방식은 특정 바이트코드 블록이 반복적으로 실행되는 시점을 감지한 후 해당 코드를 복사하여 새로운 메모리 영역에 배치하고 그 위에 직접 네이티브 코드를 패치함으로써 실행 속도를 높인다.

2.3 기존 JIT 공격 방식

De Groef 등[4]은 JIT 컴파일을 활용하는 애플리케이션에서 발생할 수 있는 코드 인젝션 공격을 방지하기 위한 보안 메커니즘인 JITSec을 제안하였다. 이 기법은 전통적인 W^X(Write XOR Execute) 정책이 사용자 공간에서 완벽히 적용되기 어렵다는 한계를 지적하고 민감한 코드와 비민감한 코드를 메모리에서 논리적으로 분리하는 구조를 도입하였다.

Chen 등[5]은 JIT Spraying 기법에 대한 방어를 목적으로 JITDefender라는 동적 보호 메커니즘을 설계 및 구현하였다. JITDefender는 JIT 컴파일러가 코드를 생성할 때 해당 메모리 영역의 실행 권한을 제거하고 오직 실행 시점에만 일시적으로 실행 권한을 부여하는 방식으로 동작하도록 하였다.

Athanasakis 등[6]은 브라우저 환경에서 JIT 컴파일러를 사용하는 가젯 없는 바이너리를 대상으로 런타임 시 동적으로 ROP(Return-Oriented Programming) 가젯을 생성할 수 있음을 보인 바 있다. 이를 통해 기존 실행 파일 내에 존재하지 않는 가젯을 JIT 영역에 생성하고, 이를 활용한 쉘코드 실행을 성공함으로써 기존의 ROP 방어 기법이 JIT 환경에서는 효과적이지 않음을 입증하였다.

이러한 선행 연구들은 공통적으로 JIT 환경이 제공하는 동적 코드 생성 및 실행 메커니즘이 공격자에게 유리한 조건을 제공하며 기존의 메모리 보호 정책만으로는 충분한 방어가 어렵다는 점을 강조하고 있다. 그러나 이들 연구는 대부분 브라우저 JIT이나 범용 JIT 환경을 대상으로 하였으며 Python의 CPython 런타임 내 Copy-and-Patch JIT 메커니즘을 구체적으로 다룬 사례는 존재하지 않는다.

따라서 본 논문에서는 CPython의 JIT 기능인 Copy-and-Patch JIT을 타겟으로 하여 Python 내부 API와 시스템 콜을 이용해 실제 메모리 권한을 변경하고, 실행 흐름을 쉘코드로 리디렉션함으로써 사용자 권한 수준에서의 임의 코드 실행 가능성을 실험적으로 입증하여 Copy-and-Patch JIT 구조의 보안상 취약 요소를 규명한다.

3. Threat Model

본 논문에서는 Copy-and-Patch JIT 환경에서 발생할 수 있는 보안 위협을 모델링한다. 제시하는 위협 모델은 공격자가 사용자 수준의 권한을 가진 상태에서 Python 실행 환경에 접근 가능한 상황을 가정하며 다음과 같은 조건 하에서 시스템이 공격에 노출될 수 있음을 전제로 한다. 첫째, 공격자는 대상 시스템 내에서 mprotect 시스템 콜과 Python Internal API의 존재를 인지하고 있으며 이를 호출할 수 있는 실행 경로를 확보하고 있다고 가정한다. 둘째, 공격자는 Python 내부 구조체 또는 Internal API를 활용하여 JIT 컴파일된 코드의 메모리 주소를 추론하거나 직접 접근할 수 있다고 가정한다. 이러한 정보를 기반으로 공격자는 JIT 메모리 영역에 대해 mprotect를 이용해 RWX(Read-Write-Execute) 권한을 부여할 수 있으며 이는 메모리 보호 정책(W^X)을 우회하고 임의 코드를 삽입 및 실행할 수 있는 기반이 된다. 본 위협 모델은 Python의 Copy-and-Patch JIT 메커니즘이 코드 복사 및 실행 권한 변경이라는 특성을 가지기 때문에 시스템 콜 수준의 제어와 사용자 공간 메모리 조작이 결합될 경우 실질적인 보안 취약점으로 연결될 수 있음을 강조한다.

4. Attack Vector

앞서 설정한 위협 모델에 따르면 공격자는 Python 내부의 특정 API 호출을 통해 JIT 컴파일러가 사용하는 메모리 주소를 추정하거나 직접 접근이 가능하며 해당 주소에 대해 시스템 콜인 mprotect를 통해 실행 권한을 부여하는 것이 가능하다. 먼저 공격자는 JIT 메모리 공간을 할당하기 위해 반복적으로 함수를 호출한다. JIT 메모리 공간이 할당된 후 CPython 내부 API 호출을 통해 내부 객체 또는 구조체의 포인터를 활용하여 JIT 컴파일된 코드가 위치한 주소를 확인하거나 Copy-and-Patch JIT이 사용하는 코드 캐시 또는 trampoline 함수의 진입 지점을 추적한다. 이후 ctypes 모듈이나 C 확장 모듈을 이용해 시스템 콜 인터페이스에 접근하고 타겟 메모리 영역에 RWX 권한을 부여하기 위해 mprotect 시스템 콜을 호출한다. 이때 주목할 점은 Copy-and-Patch JIT이 성능 최적화를 위해 복사한 네이티브 코드는 일반적으로 힙 또는 mmap된 메모리 상에 위치하며 기본적으로는 RW 권한만을 갖는다. 그러나 mprotect를 통해 해당 영역을 실행 가능하게 변경하는 순간 이는 악성 페이로드의 적재 및 실행이 가능한 위험한 실행 공간으로 전환된다. 문제는 mprotect 시스템 콜 자체가 커널 레벨에서의 명시적 권한 제어 없이 사용자 프로세스 내부에서 비교적 자유롭게 호출 가능하다는 점이다. 이는 W^X 기반의 메모리 보호 정책이 이론적으로는 안전성을 제공하지만 실질적으로는 Python과 같은 인터프리터 환경에서 충분한 보호를 보장하지 못함을 의미한다. 즉, 실행 중 메모리 권한을 변경하는 것을 제한할 기술적 수단이 부재한 상황에서 공격자는 정당한 JIT 실행 경로를 가장해 악의적인 코드 실행으로 흐름을 유도할 수 있다. 공격자는 메모리 공간에 ROP chain 또는 쉘코드를 삽입하고, trampoline을 재작성하거나 컨트롤 플로우를 조작하여 해당 코드가 실행되도록 유도할 수 있다. 이러한 공격 흐름은 단순한 메모리 권한 변경으로부터 시작하여 실행 흐름 제어, 코드 삽입, 최종적으로 임의 코드 실행까지 이어지는 다단계 구조를 가지며 Python 실행 환경이 신뢰된 인터프리터로 운영되는 시스템에서도 잠재적 보안 취약점을 야기할 수 있음을 보여준다.

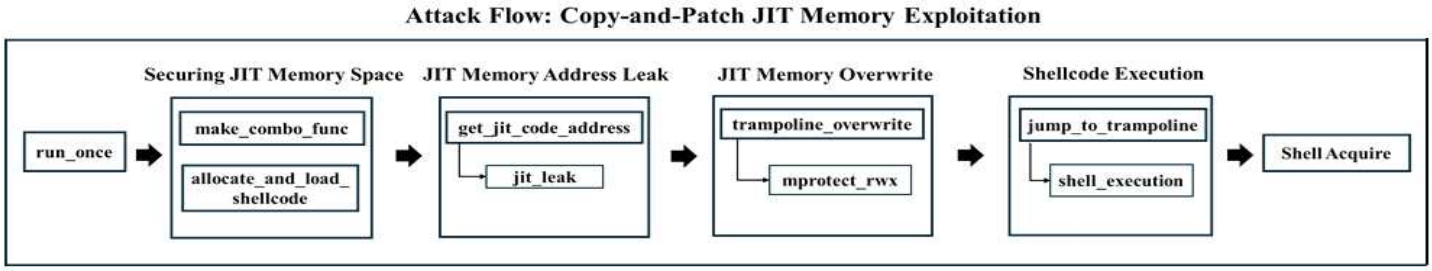

그림 1. Copy-and-Patch JIT 공격 흐름도

```
[*] code object @ 0xffffa3f05540
[*] executor found at offset 0
[*] executor @ 0xaaab1f98f820
[*] executor->jit_code @ 0xffffa3976000
[*] executor->jit_size @ 40960
#
```

그림 2. 쉘 획득 결과 화면

5. Proof of Concept (PoC)

본 논문에서는 앞서 제시한 공격 벡터를 기반으로 실제 환경에서 임의 코드 실행 및 쉘 획득이 가능한지를 검증하였다. 실험 환경은 aarch64, Ubuntu 22.04.05 LTS, Python 3.14 환경에서 수행되었으며 Copy-and-Patch JIT 환경을 구성한 후 Python 내부 API와 시스템 콜을 활용하여 공격 시나리오를 구현하였다. 그림 1은 공격 과정을 단계별로 도식화한 것으로 각 단계는 JIT 메모리 확보, 메모리 주소 유출, 권한 변경, 쉘코드 실행 순으로 구성된다. 공격은 run_once 함수 호출을 통해 시작되며 이 함수는 JIT 메모리 공간을 확보하기 위해 make_combo_func 함수를 호출한다. 해당 함수는 반복 코드 호출을 통해 JIT 컴파일러가 코드를 생성하도록 유도한다. 이 과정을 통해 JIT 컴파일러에 의해 생성되는 네이티브 코드 블록이 힙 혹은 mmap된 메모리 공간에 배치되며 공격자는 해당 공간에 사전에 정의된 쉘코드를 할당하고 로드한다. 이후 공격자는 get_jit_code_address 함수를 통해 해당 JIT 메모리 영역의 시작 주소를 유출한다. 해당 함수는 외부 모듈인 jit_leak 함수를 활용하여 JIT 컴파일러가 생성한 코드 블록의 실제 메모리 주소를 식별한다. 메모리 주소가 확보되면 다음 단계로 trampoline_overwrite 함수를 호출해 JIT 메모리 주소에 trampoline 코드를 삽입하고 시스템 콜 mprotect를 이용하여 해당 메모리 블록에 RWX 권한을 부여한다. 최종적으로 jump_to_trampoline 함수 호출을 통해 공격자는 trampoline을 실행하여 제어 흐름을 쉘코드가 위치한 메모리로 리디렉션 시키고 이를 실행함으로써 '/bin/sh' 명령어 실행을 성공시킨다. 그림 2는 공격이 성공적으로 수행된 후 대상 시스템 상에서 쉘이 활성화된 결과를 보여준다. 본 실험을 통해 제안한 공격이 실제 시스템 상에서도 재현 가능하며 Copy-and-Patch JIT 환경이 코드 실행 경로로 악용될 수 있음을 입증하였다.

6. 결론 및 향후 연구

본 논문은 Python 런타임 환경에서 동작하는 Copy-and-Patch JIT 메커니즘의 구조적 특성과 보안 취약점을 분석하고 이를 활용한 공격 시나리오를 실제 시스템 환경에서 검증하였다. 제안한 공격은 정형화된 위협 모델과 공격 벡터를 기반으로 설계되었으며 실험을 통해 Proof of Concept(PoC)를 구현함으로써 공격자의 입장에서 JIT 메모리 영역의 제어 권한을 확보하고 최종적으로 쉘코드 실행까지 도달할 수 있음을 입증하였다. ARM 아키텍처의 경우 고정된 명령어 길이를 가지므로 ROP 및 JIT Spraying과 같은 공격이 비교적 용이하게 적용될 수 있으며 이에 따라 보안 대책의 마련이 시급하다. 반면, Intel x86 아키텍처는 가변 길이 명령어 구조로 인해 동일한 공격 기법의 적용 가능성에 대한 기술적 검증이 필요하다. 이에 따라 향후 연구에서는 동일한 공격 흐름이 Intel x86 기반 환경에서도 재현 가능한지를 확인하고 각 아키텍처 특성에 최적화된 ROP 및 메모리 조작 기법을 탐색할 것이다. 이를 통해 Python 실행 환경 전반에서의 JIT 기반 보안 취약점을 심층적으로 분석하고, 보다 보편적이며 효과적인 방어 기법 및 패치 적용의 필요성을 공동체에 제안하는 것을 목표로 한다.

참 고 문 헌

[1] H. Xu and F. Kjolstad, "Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode," in Proceedings of the ACM on Programming Languages, Vol. 5, OOPSLA, Chicago, IL, USA, 2021
[2] https://docs.python.org/3/whatsnew/3.13.html
[3] https://tonybaloney.github.io/posts/python-gets-a-jit.html
[4] De Groef, Willem, et al. "Jitsec: Just-in-time security for code injection attacks." Benelux Workshop on Information and System Security (WISSEC 2010)
[5] Chen, Ping, et al. "JITDefender: A defense against JIT spraying attacks." Future Challenges in Security and Privacy for Academia and Industry: 26th IFIP TC 11 International Information Security Conference, 2011
[6] Athanasakis, Michalis, et al. "The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines." NDSS. 2015.