

Amadey의 가면 벗기기

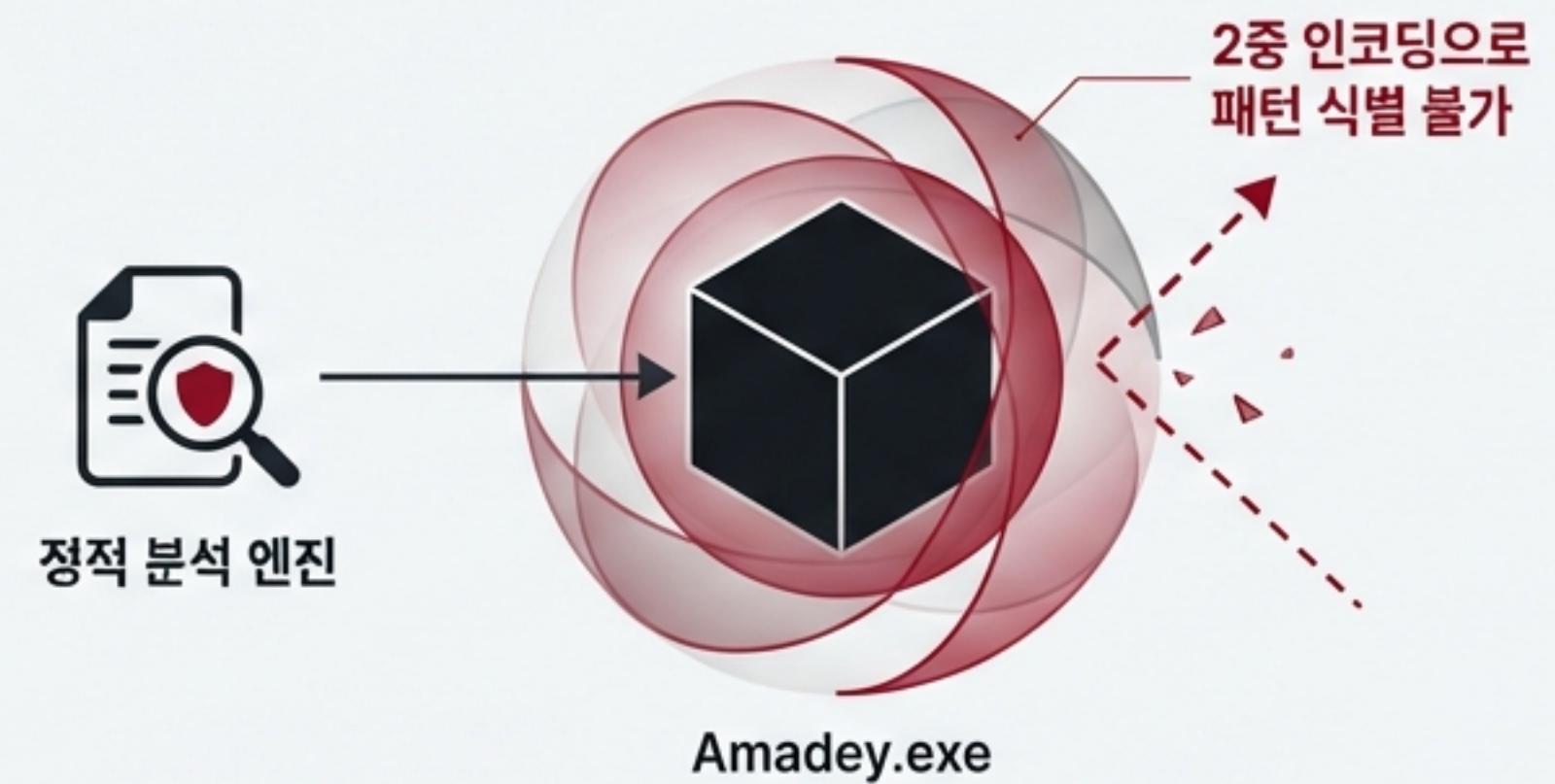


파일의 흔적이 아닌, 메모리의 진실을 추적하는 방법

교묘한 적: 정적 탐지를 무력화하는 Amadey

정적 탐지는 디스크 상의 파일 서명을 분석하지만,
Amadey와 같은 최신 악성코드는 이를 쉽게 우회합니다.

- **핵심 우회 기법:** Amadey는 페이로드 내 주요
문자열을 **2중으로 인코딩(사용자 정의 인코딩 +
Base64)**하여 정적 분석을 극도로 어렵게 만듭니다.
- **결과:** 패커(packer)나 암호화로 감싸진 변종 악성코드는
기존의 파일 해시나 문자열 서명 기반 탐지를
무력화시킵니다. 이는 **MaaS(Malware-as-a-Service)**
형태로 빠르게 유포되는 Amadey 변종 탐지에
치명적인 약점입니다.



두 가지 접근법: 파일 vs. 메모리

탐지 방식	장점	단점	공격자 회피 난이도
정적 탐지 (Static) 파일 기준	빠른 검사, 알려진 서명에 대한 높은 정확도	패킹/암호화에 취약, 신규 변종 누락 위험	낮음 (Low) 간단한 패커나 인코딩으로 우회 가능
메모리 기반 탐지 (Memory-based) 프로세스 기준	난독화 우회 (복호화된 실제 코드 탐지), 실행 행위 기반 탐지 (파일리스 공격 대응)	실시간 스캔 오버헤드 발생 가능, 전문 분석 도구 필요	높음 (High) 핵심 코드 로직 자체를 변경해야 함

정적 탐지는 "파일이 어떻게 생겼는가"를 보지만, 메모리 탐지는 "프로세스가 실제 무엇을 하는가"를 파악합니다.

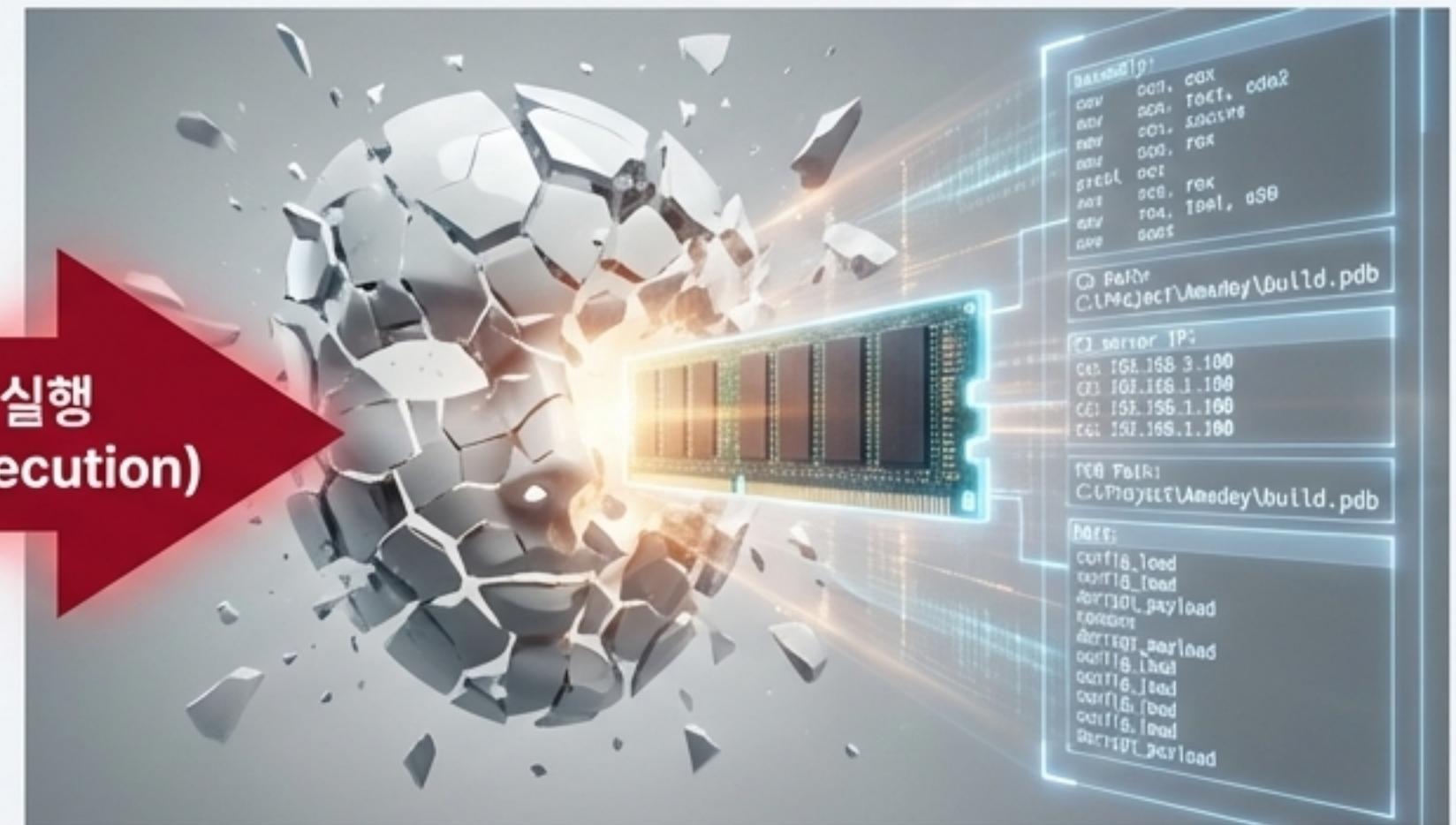
패러다임의 전환: "런타임은 거짓말하지 않는다"

디스크 위 (On Disk)



프로세스 실행
(Process Execution)

메모리 속 (In Memory)



디스크 위 (On Disk)

메모리 속 (In Memory)

- 아무리 정교하게 포장된 악성코드라도, 실행되는 순간에는 반드시 메모리상에서 자신의 실제 코드와 데이터를 복호화해야 합니다.
- 공격자는 디스크 상의 파일 모습은 얼마든지 위장할 수 있지만, 실행을 위해 메모리에 드러난 자신의 본질은 숨길 수 없습니다.
- 우리의 목표는 위장된 파일이 아닌, 메모리에 펼쳐진 이 “**진실**”을 포착하는 것입니다.

적의 약점: 실행 불변 특징 (Execution Invariants)

실행 불변 특징(Invariant)이란, 악성코드가 변종/난독화에도 불구하고 핵심 기능을 수행하기 위해 어쩔 수 없이 메모리에 남기는 고유한 코드나 데이터 패턴을 의미합니다. 이는 공격자가 쉽게 바꿀 수 없는 알고리즘의 지문과 같습니다.



Amadey

메모리에서 발견되는 32글자 길이의 고정 HEX
복호화 키, 개발 경로 문자열
\Amadey\Release\Amadey.pdb

RedLine Stealer

브라우저 DB 경로 cookies.sqlite나 기능명
ChromeGetRoamingName,
DownloadAndExecuteUpdate 같은 문자열

FormBook

프로세스 인젝션을 위해 필수적인 API 호출 순서
(e.g., VirtualAllocEx → WriteProcessMemory
→ ResumeThread)

공격자의 딜레마: 왜 불변 특징은 쉽게 사라지지 않는가?

1. 기능 유지의 제약 (Functional Constraints)

악성 행위(정보 탈취, C2 통신 등)의 목표는 동일하므로, 핵심 기능을 구현하는 코드 구조나 데이터는 필연적으로 남게 됩니다.



2. 회피를 위한 막대한 비용 (Prohibitive Evasion Cost)

Invariant를 제거하려면 아래와 같은 고비용 기술이 필요합니다.

- **메타모픽(Metamorphic) 엔진:** 실행 시마다 코드를 재조합. 개발 난이도가 극도로 높고 성능 저하 및 안정성 문제를 유발합니다.
- **실시간 암/복호화:** 필요한 코드만 잠시 복호화 후 다시 암호화. 심각한 CPU 오버헤드를 발생시킵니다.
- **가상머신(VM) 기반 실행:** 커스텀 인터프리터 방식. VM 개발/유지보수 비용이 크고, 인터프리터 자체가 새로운 탐지 시그니처가 됩니다.

Invariant 제거는 공격자에게 득보다 실이 큽니다.

특히 MaaS 형태의 악성코드는 개발 효율성을 위해 코어 로직을 재사용하는 경향이 강합니다.

사냥의 시작: YARA를 이용한 불변 특징 탐지

- YARA는 텍스트/바이너리 패턴을 정의하여 악성코드 군을 식별하는 강력한 도구입니다.
- 파일뿐만 아니라, 실행 중인 프로세스 메모리나 메모리 덤프를 직접 스캔하여 난독화 너머의 실제 페이로드를 탐지할 수 있습니다.
- 해시 매칭과 달리 일반화된 패턴을 사용하므로, 수많은 폴리모픽 변종에 대응하는 **불변 특징(Invariant)** 탐지에 **최적화**되어 있습니다.

```
rule Detect_Amadey_Invariant_InMemory {  
    meta:  
        description = "Detects Amadey PDB path or key patterns in memory"  
    strings:  
        // Evidence 1: Development artifact  
        $pdb = "\\\Amadey\\\\Release\\\\Amadey.pdb" ascii wide ← 개발 경로 문자열  
        // Evidence 2: Unique code sequence  
        $seq_0 = { 89 45 F4 83 7D F4 08 74 4F } ← 고유 코드 실행 흐름  
    condition:  
        // Found in a PE file loaded in memory  
        uint16(0) == 0x5A4D and any of them  
}
```

하나의 패턴, 여러 개의 표적: 악성코드 패밀리 간 메모리 수렴성

서로 다른 악성코드들도 결국 메모리상에서는 유사한 형태나 동작(수렴성)을 보입니다.

이는 각 패밀리가 수행하는 고유한 악성 행위가 공통된 기술적 흔적을 남기기 때문입니다.



Amadey (Loader)

Invariant

문자열 복호화 루틴, C2 통신 스레드
생성 코드.



RedLine (Stealer)

Invariant

브라우저 프로파일, 암호화폐 지갑 관련
평문 문자열.



FormBook (Injector)

Invariant

다층 암호화 후 최종 단계의 프로세스 인젝션
API 호출 시퀀스.



SmokeLoader (Loader)

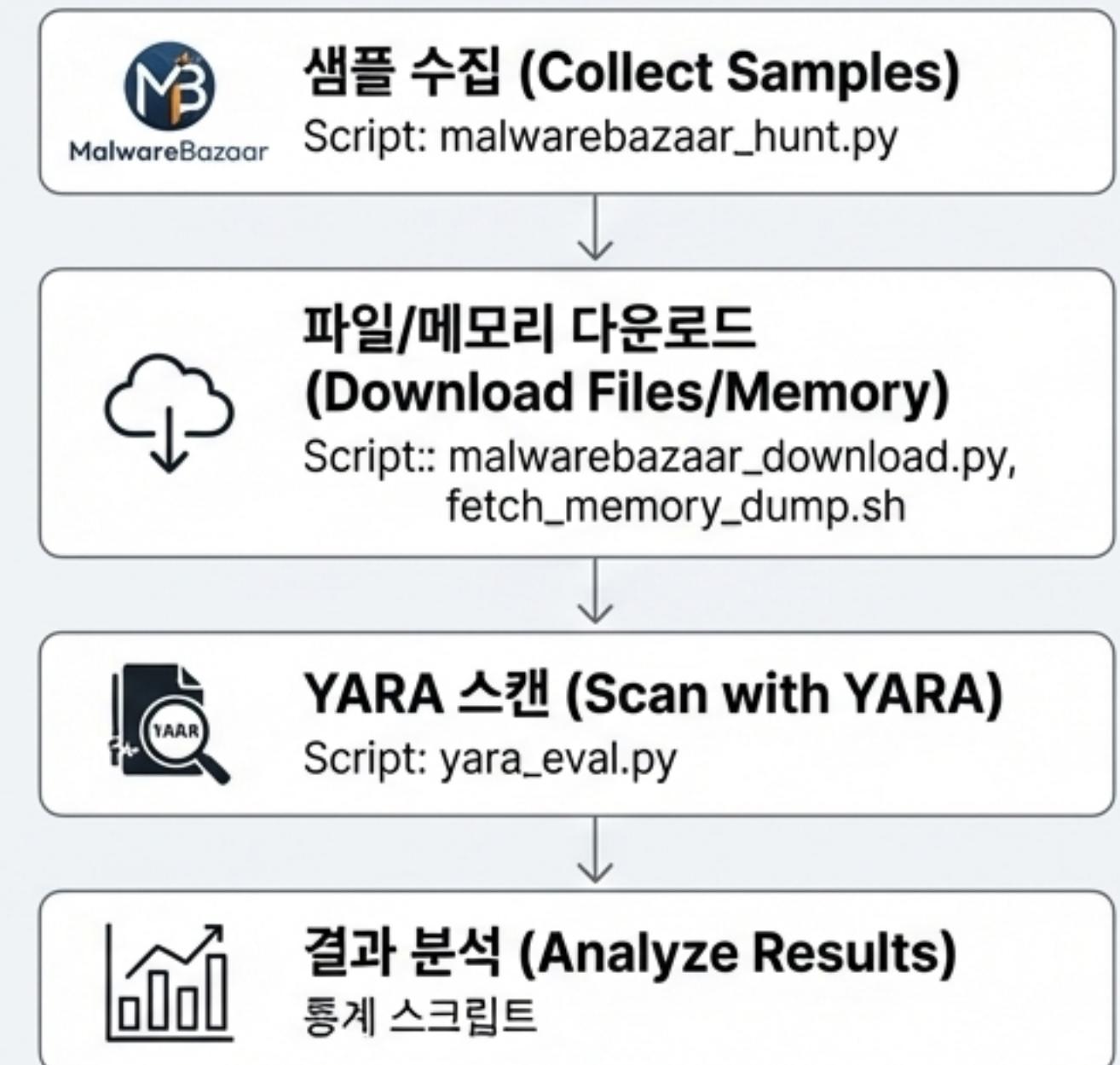
Invariant

`explorer.exe` 등 정상 프로세스에
인젝션된 비정상 코드 영역.

가설에서 실증으로: 반복 가능한 데이터 기반 실험 환경 `apt`

Introduction to `apt` (Amadey Pipeline Tools)

- 악성코드 탐지율을 직접 계산하고 실증하기 위해 설계된 오픈소스 도구 모음입니다.
- 보안 솔루션의 방해를 피하기 위해 **모든 프로세스를 Docker 기반 격리 환경**에서 안전하게 수행합니다.
- 이 환경을 통해 "**메모리 기반 탐지가 정적 탐지보다 우월하다**"는 가설을 정량적으로 검증했습니다.



github.com/windshock/apt

`apt` 파이프라인 워크플로우

Step 1: 해시 목록 준비 (Prepare Hash List)

```
# Amadey 태그로 100개 샘플 해시 수집  
bash scripts/apt_docker.sh python3 malwarebazaar_hunt.py --tag amadey --limit 100
```

Step 2: 악성코드 샘플 다운로드 (Download Malware Samples)

```
# 해시 목록 기반으로 파일 다운로드 및 압축 해제  
bash scripts/apt_docker.sh python3 malwarebazaar_download.py --file /data/amadey_100_hashes.txt
```

Step 3: 메모리 덤프 다운로드 (Download Memory Dumps)

```
# Hybrid-Analysis에서 메모리 덤프 다운로드  
bash scripts/apt_docker.sh bash fetch_memory_dump.sh --sha256-list /data/amadey_100_hashes.txt
```

Step 4: YARA 스캔 및 결과 집계 (Scan with YARA & Aggregate Results)

```
# 다운로드된 파일 스캔  
bash scripts/apt_docker.sh python3 scripts/yara_eval.py --rules ... --target /data/unzip_amd_100  
  
# 메모리 덤프 스캔 (압축 해제 후)  
bash scripts/apt_docker.sh python3 scripts/yara_eval.py --rules ... --target /data/ha.dumps_unz
```

모든 과정은 스크립트화되어 있으며, Docker 안에서 안전하고 반복적으로 실행할 수 있습니다.

결정적 증거: 데이터가 말해주는 진실

Amadey 샘플 100개를 대상으로 `apt` 파이프라인을 통해 동일한 YARA 룰셋(Yaraify YaraHub)을 적용한 실험 결과입니다.

71%

파일 기반 탐지율 (File-based Detection)

100개의 원본 파일 샘플 중 71개 탐지

95.24%

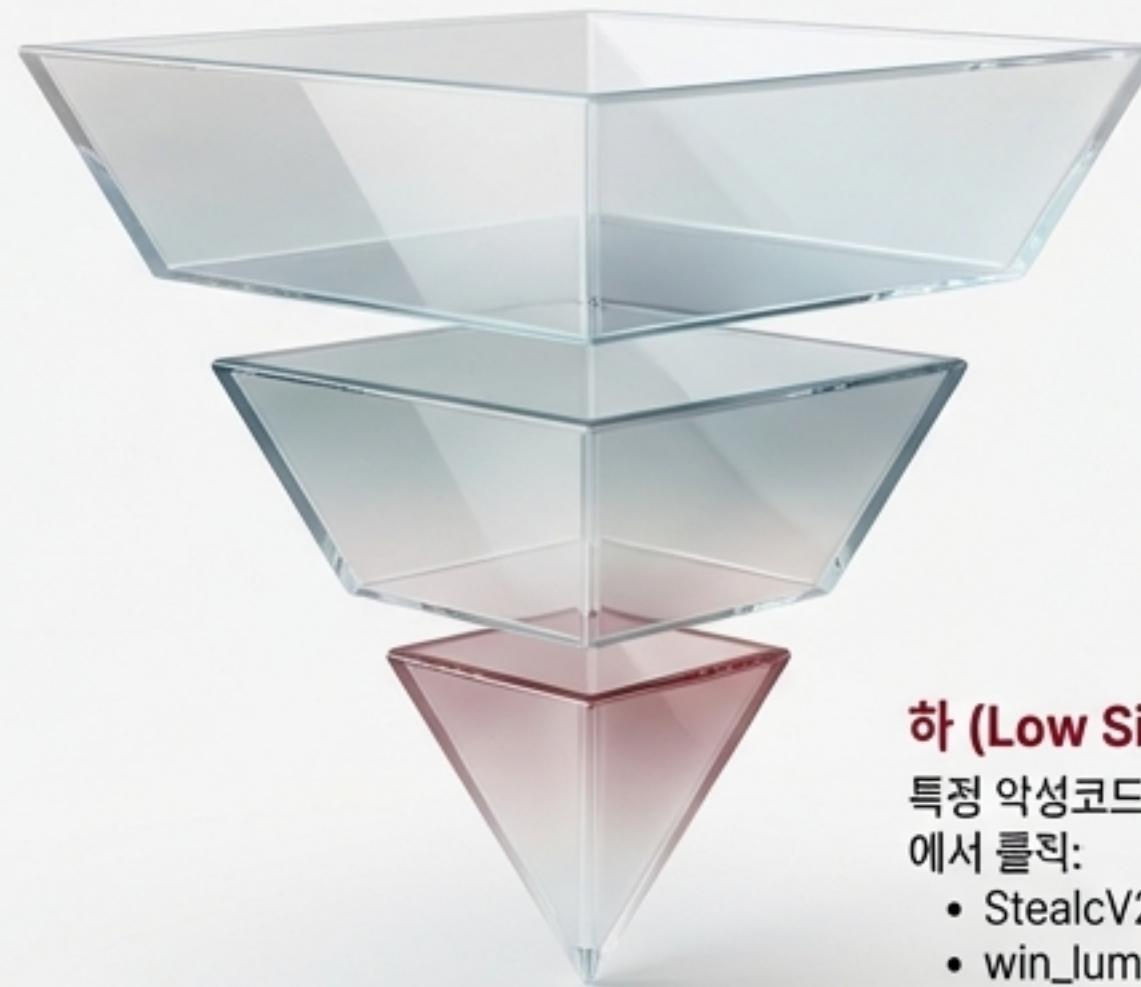
메모리 기반 탐지율 (Memory-based Detection)

분석 가능한 63개의 메모리 덤프 중 60개 탐지



탐지를 넘어 프로파일링으로: 메모리 신호의 이해

YaraHub 룰셋을 활용하면 단순 탐지를 넘어, 메모리의 "성격"을 프로파일링할 수 있습니다.
모든 룰이 같은 의미를 갖는 것은 아닙니다.



상 (High Signal) - 행위의 단서 (Behavioral Clues)

로더/언패커 등에서 공통적으로 나타나는 일반적인 악성 행위의 흔적.

- meth_get_eip
- pe_detect_tls_callbacks
- DetectEncryptedVariants

중 (Mid Signal) - 도구의 흔적 (Tooling Artifacts)

공격 프레임워크(예: Cobalt Strike)나 특정 캠페인에서 사용된 도구의 흔적.
에서 흔적:

- cobalt_strike_tmp01925d3f
- RANSOMWARE

하 (Low Signal) - 패밀리 확증 (Family Confirmation)

특정 악성코드 패밀리를 높은 신뢰도로 확증할 수 있는 고유한 시그니처.
에서 를:

- StealcV2
- win_lumma_generic

메모리 분석은 "이것이 악성코드인가?"라는 질문뿐만 아니라,
"이 악성코드는 어떤 종류의 행위를 하는가?"에 대한 깊이 있는 답변을 제공합니다.

이제 당신의 차례입니다: `apt` 툴킷으로 직접 확인하십시오

- 본 발표에 사용된 모든 워크플로우와 스크립트는 공개된 GitHub 리포지토리에서 사용할 수 있습니다.
- 직접 이 실험을 재현하고, 탐지율을 검증하고, 여러분의 YARA 룰을 테스트해보십시오.
- **주요 기능:**
 - 악성코드 샘플 및 메모리 덤프의 자동 수집
 - 안전한 Docker 격리 환경 내에서 YARA 스캔 수행
 - 탐지 결과 및 통계 자동 생성



github.com/windshock/apt

핵심 요약: 조사에서 전략으로



1. 정적 탐지는 한계에 도달했습니다 (Static Detection Has Reached Its Limit)

최신 악성코드는 패킹과 난독화로 기존 시그니처 기반 방어를 쉽게 우회합니다.



2. 메모리는 악성코드의 본질을 드러냅니다 (Memory Reveals a Malware's Essence)

실행 불변 특징(Execution Invariants)은 난독화 너머의 진실을 보여주는 신뢰도 높은 증거입니다.



3. 견고하고 지속 가능한 방어 전략입니다 (A Robust and Sustainable Strategy)

공격자 입장에서 메모리 불변 특징을 회피하는 것은 기술적, 비용적으로 매우 어렵습니다.



4. 누구나 검증하고 활용할 수 있습니다 (Accessible for Verification and Use)

'apt'와 YARA 같은 오픈소스 도구를 통해 이 접근법을 즉시 현업에 적용하고 검증할 수 있습니다.



> 파일은 위장할 수 있지만,
> 메모리는 거짓말하지 않습니다.

(Files can wear a disguise, but memory does not lie.)