Rampart: Protecting Web Applications from CPU-Exhaustion Denial-of-Service Attacks

Wei Meng[†], Chenxiong Qian[‡], Shuang Hao[*], Kevin Borgolte[§]
Giovanni Vigna[§], Christopher Kruegel[§], Wenke Lee[‡]

[†]Chinese University of Hong Kong
[‡]Georgia Institute of Technology
[*]University of Texas at Dallas
[§]University of California, Santa Barbara

Outline

- Background & Motivation
- Rampart
- Performance Evaluation
- Mitigation Evaluation

Denial-of-Service (DoS) Attacks

- A class of attacks on availability
  - Keeping users from using a certain computing service
- Two types of DoS attacks
  - Program flaw
    - Supplying an input that can crash the target application or system
  - Resource exhaustion (focus of this work)
    - Requesting a significant amount of computing resources, e.g., CPU, memory, disk, network connections

Distributed DoS (DDoS) Attacks

- Attackers need to send traffic at a rate greater than the bottleneck processing capacity of the target system
  - DoS attacks are usually launched by flooding the target system with excessive traffic to impair the target's availability
- DDoS attackers send the traffic from more than one single source
  - E.g., crafting requests from thousands of bots using many IP addresses
  - Higher bandwidth + more difficult to prevent
- Amplification techniques (e.g., DNS reflection) can be used in DDoS attacks to further increase the bandwidth of the attack traffic

Low-volume Sophisticated DoS Attacks

- Attackers need to send traffic at a rate greater than the bottleneck processing capacity of the target system
  - What if I do not have control over thousands of machines?
- Low-volume sophisticated DoS attacks
  - Less but much more intense (computationally expensive) attack traffic
  - E.g., requesting the server to compute a hash for millions of times

CVE-2014-9034

Description

wp-includes/class-phpass.php in WordPress before 3.7.5, 3.8.x before 3.8.5, 3.9.x before 3.9.3, and 4.x before 4.0.1 allows remote attackers to cause a denial of service (CPU consumption) via a long password that is improperly handled during hashing, a similar issue to CVE-2014-9016.

```
function HashPassword($password)
{
  $random = '';

  if (CRYPT_BLOWFISH == 1 && !$this->portable_hashes) {
    $random = $this->get_random_bytes(16);
    $hash =
        crypt($password, $this->gensalt_blowfish($random));
    if (strlen($hash) == 60)
      return $hash;
  }

  /* ... */

  if (strlen($random) < 6)
    $random = $this->get_random_bytes(6);
  $hash =
      $this->crypt_private($password,
      $this->gensalt_private($random));

  if (strlen($hash) == 34)
    return $hash;

  return '*';
}
```

https://github.com/WordPress/WordPress/blob/3.6-branch/wp-includes/class-phpass.php

```
string crypt ( string $str [, string $salt ] )
```

crypt() will return a hashed string using the standard Unix DES-based algorithm or alternative algorithms that may be available on the system.

Fix

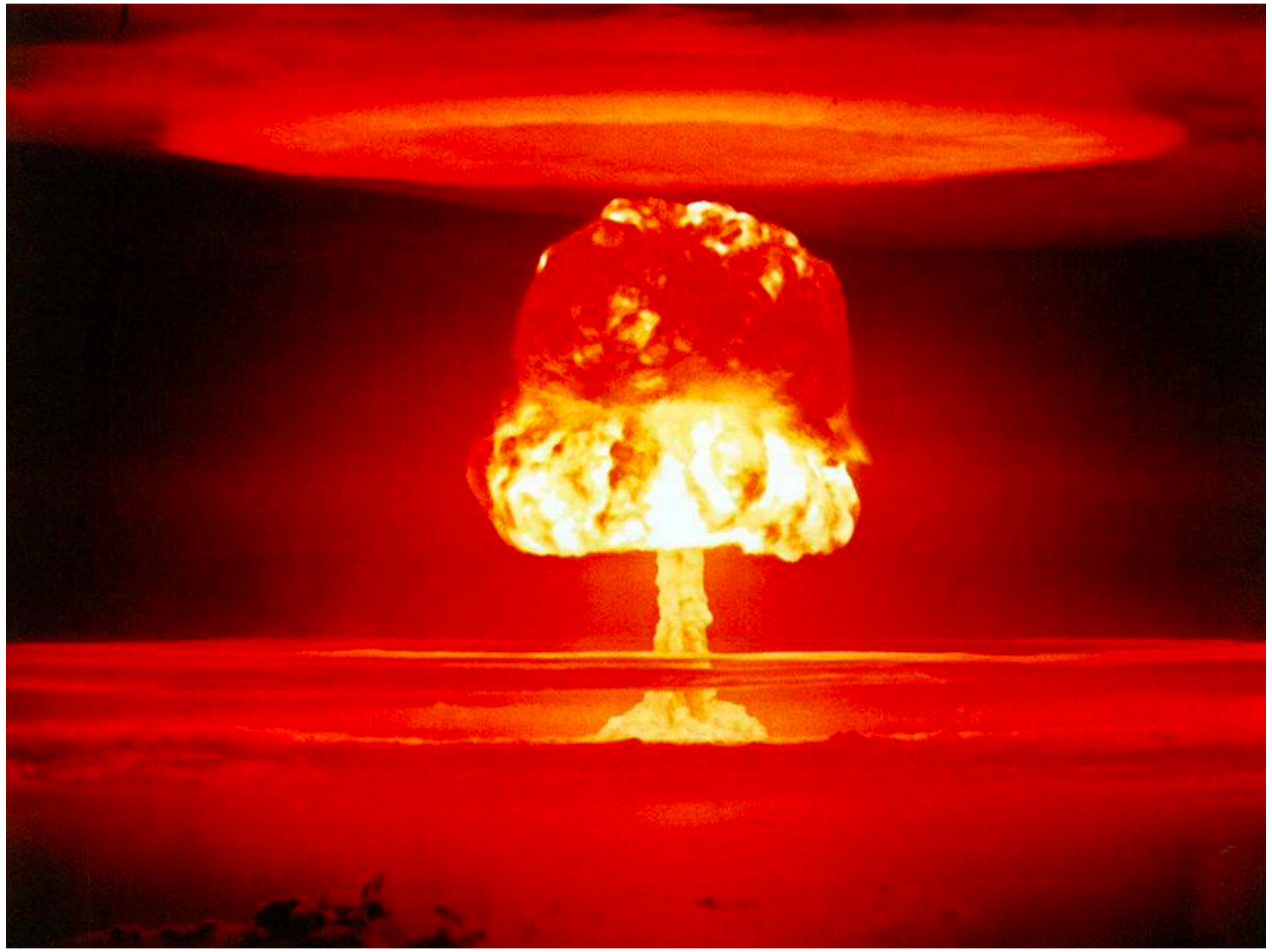
```
function HashPassword($password)
{
  if ( strlen( $password ) > 4096 ) {
    return '*';
  }

  $random = '';

  if (CRYPT_BLOWFISH == 1 && !$this->portable_hashes) {
    $random = $this->get_random_bytes(16);
    $hash =
        crypt($password, $this->gensalt_blowfish($random));
    if (strlen($hash) == 60)
      return $hash;
  }

  /* ... */

  return '*';
}
```

https://github.com/WordPress/WordPress/blob/3.7-branch/wp-includes/class-phpass.php

Conventional DDoS Attacks

https://www.smithsonianmag.com/history/seventy-years-world-war-two-thousands-tons-unexploded-bombs-germany-180957680/

Sophisticated DoS Attacks



https://www.smithsonianmag.com/smart-news/25-years-us-special-forces-carried-miniature-nukes-their-backs-180949700/

Goals

- Protecting the back end of web applications from
- Low-volume sophisticated CPU-exhaustion DoS attacks
- While limiting impact caused by false-positives

Outline

- Background & Motivation
- Rampart
- Performance Evaluation
- Mitigation Evaluation

Threat Model

- The back-end of a web application is vulnerable against CPU-exhaustion DoS attacks
- The goal of an attacker is to occupy all available CPU resources of the server
- The attacker sends attack payload through normal HTTP requests at a low rate
- The attack requests cannot be easily distinguished from legitimate requests through statistical features
- The attacker does not flood the server with numerous requests

Approach

- Web application CPU usage modeling through context-aware function-level program profiling

- Attack detection using statistical execution model

- Probabilistic request termination

- Exploratory attack request blocking

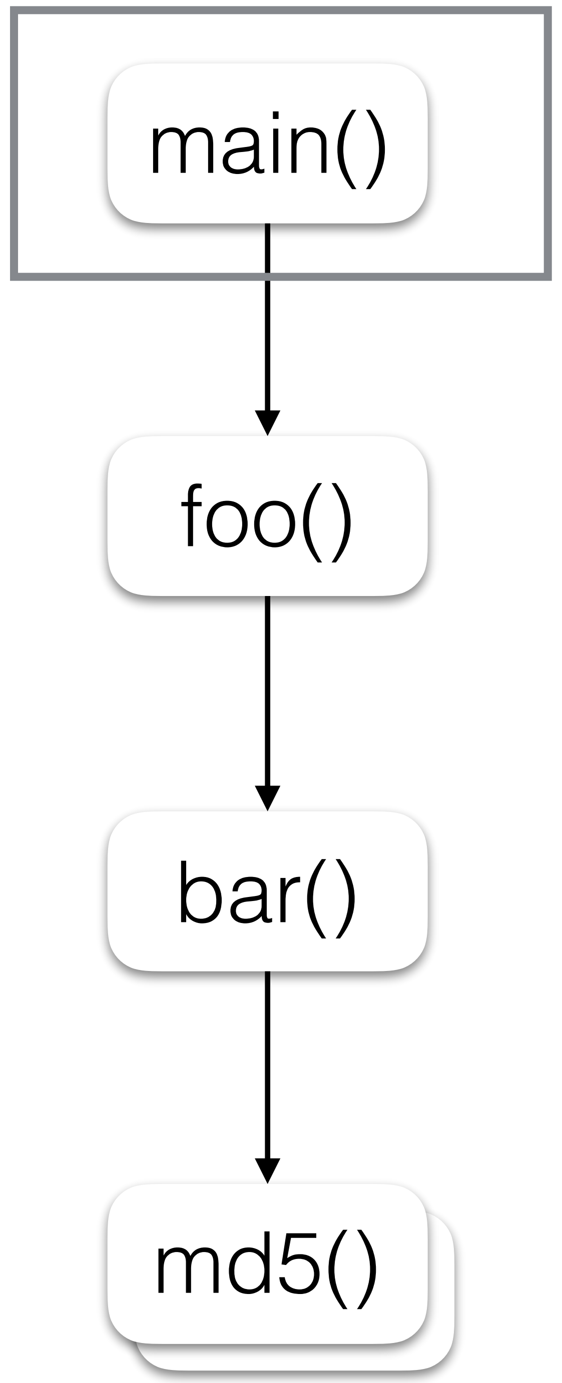
- Performance optimizations

Web Application CPU Usage Modeling

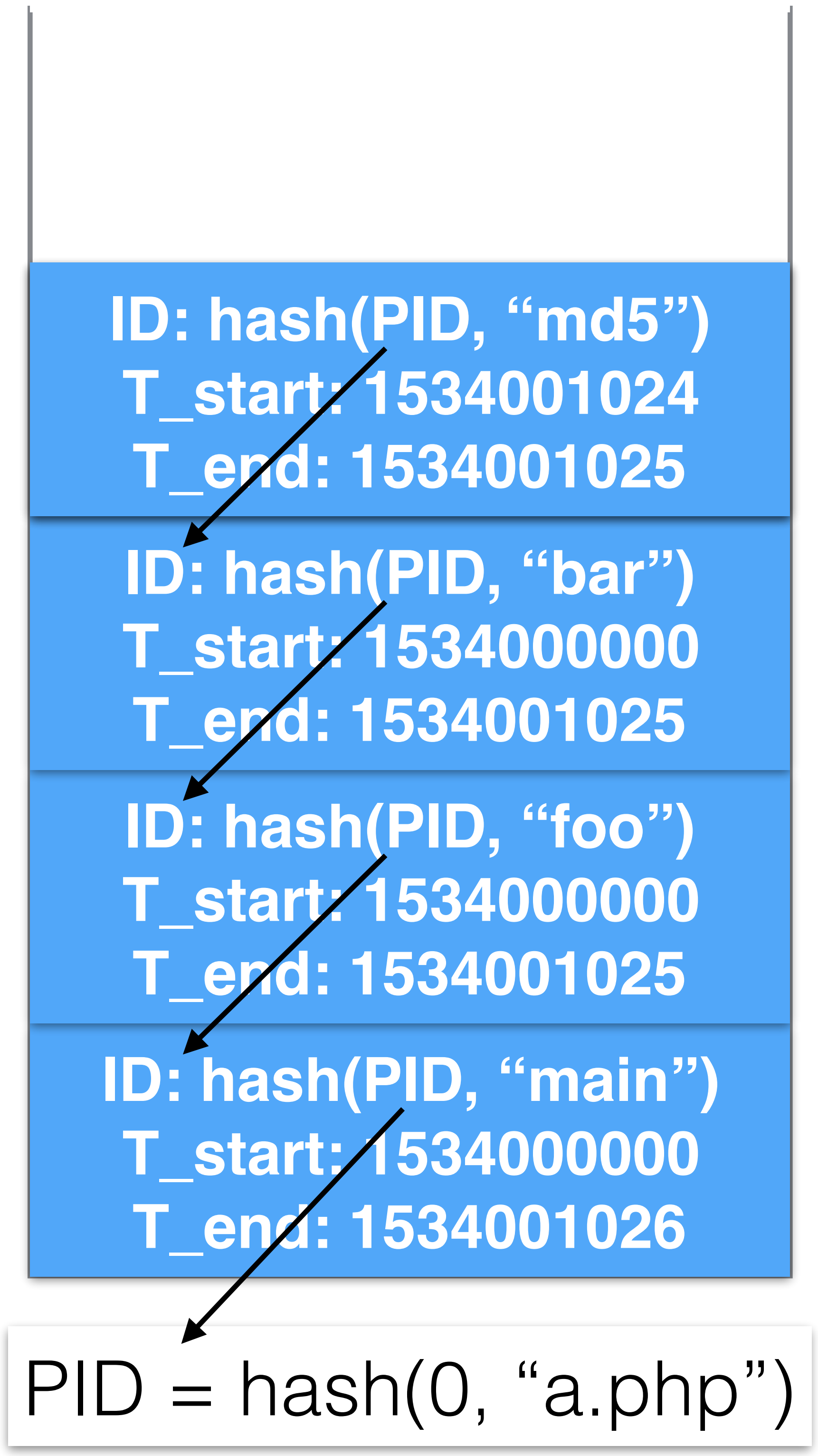
```
1.require_once 'lib.php';
2.function foo() {
3.  return bar(1);
4.}
5.$r = foo();
```
a.php

```
1.function bar($f) {
2.  $val = "a";
3.  if ( $f > 0 ) {
4.    for ( $i = 0; $i < 1024; $i++ ) {
5.      $val = md5($val, TRUE);
6.    }
7.  }
8.  return $val;
9.}
```
lib.php

```
1.require_once 'lib.php';
2.function foo() {
3.  return bar(0);
4.}
5.$r = foo();
6.$x = bar(1);
```
b.php



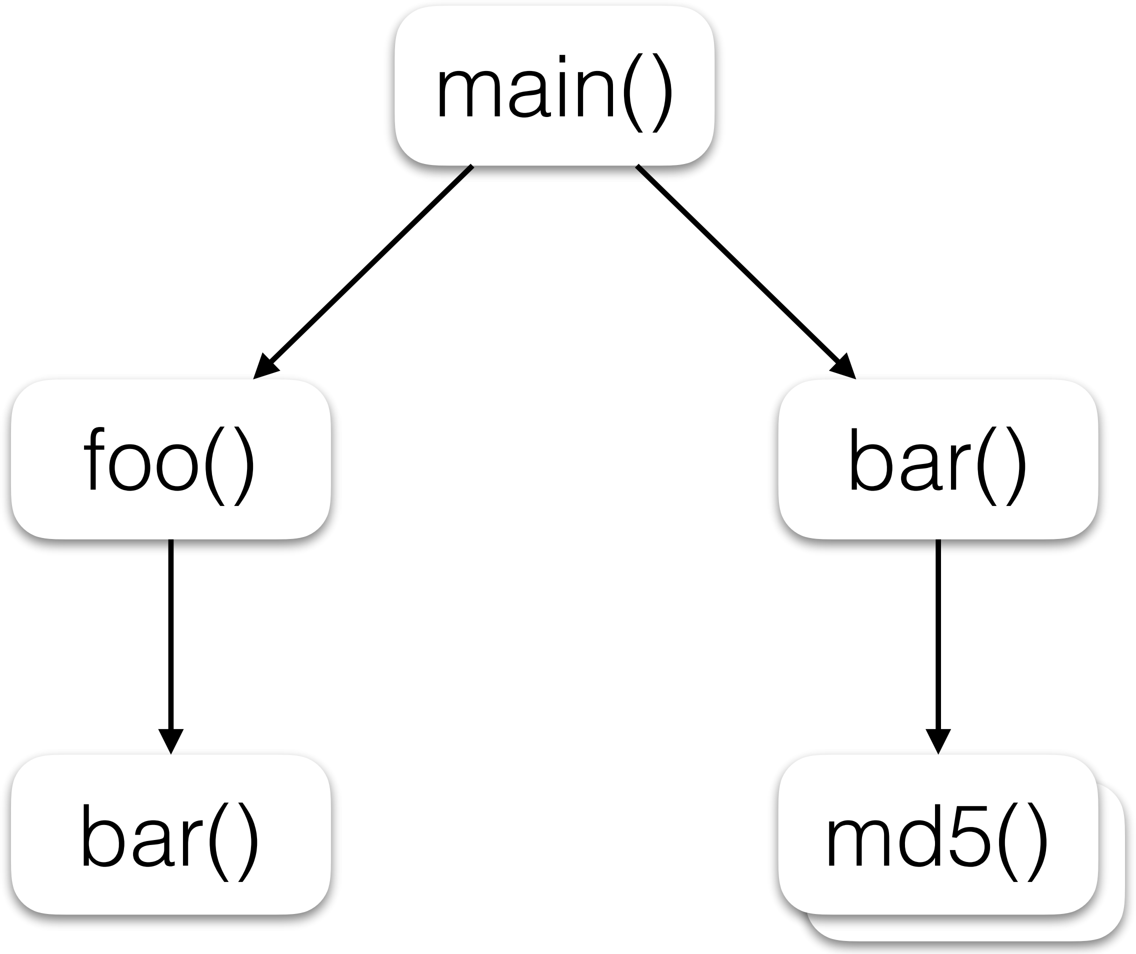
Call Stack



PID stands for the ID of the parent frame

Function Execution Records

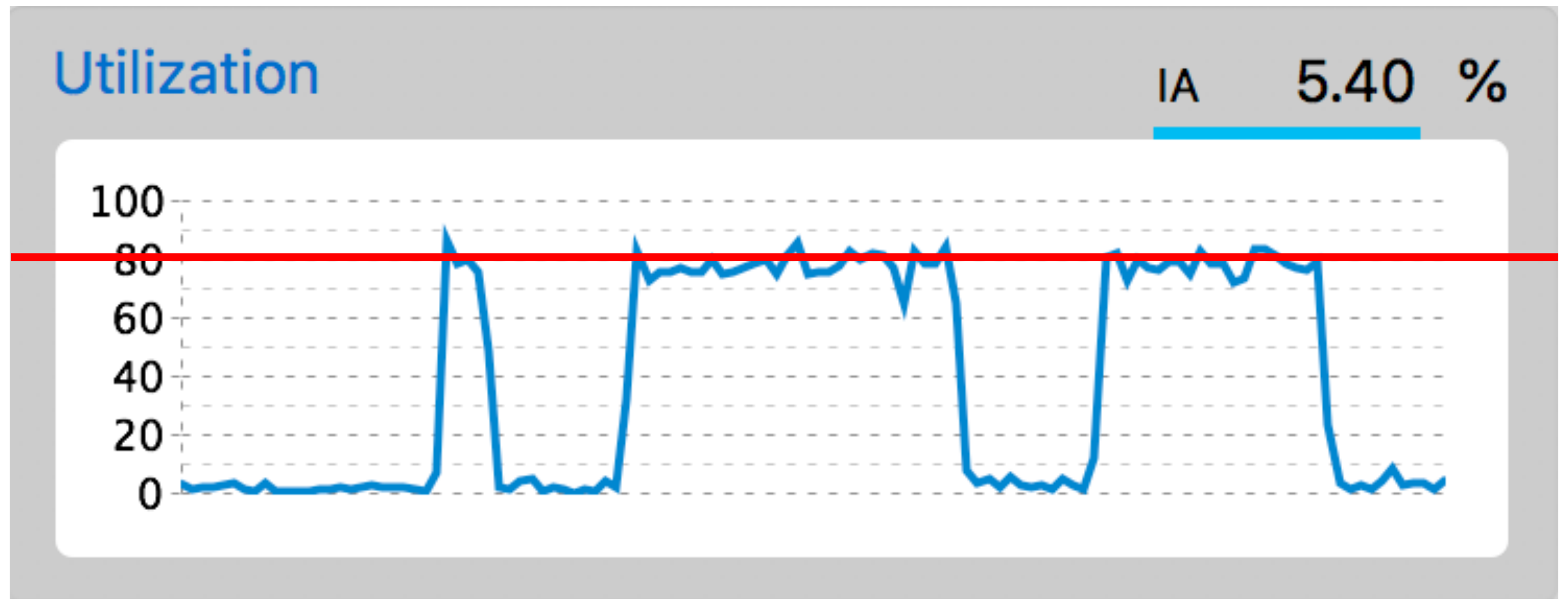
Function	CPU time measurements
a7f2943c	1026
1c39686a	1025
8009ece6	1025
3825111	1 1.1 0.9 1 … 1



The measured time is CPU time not wall-clock time

CPU-Exhaustion DoS Attack Detection

- How to detect CPU-exhaustion DoS?

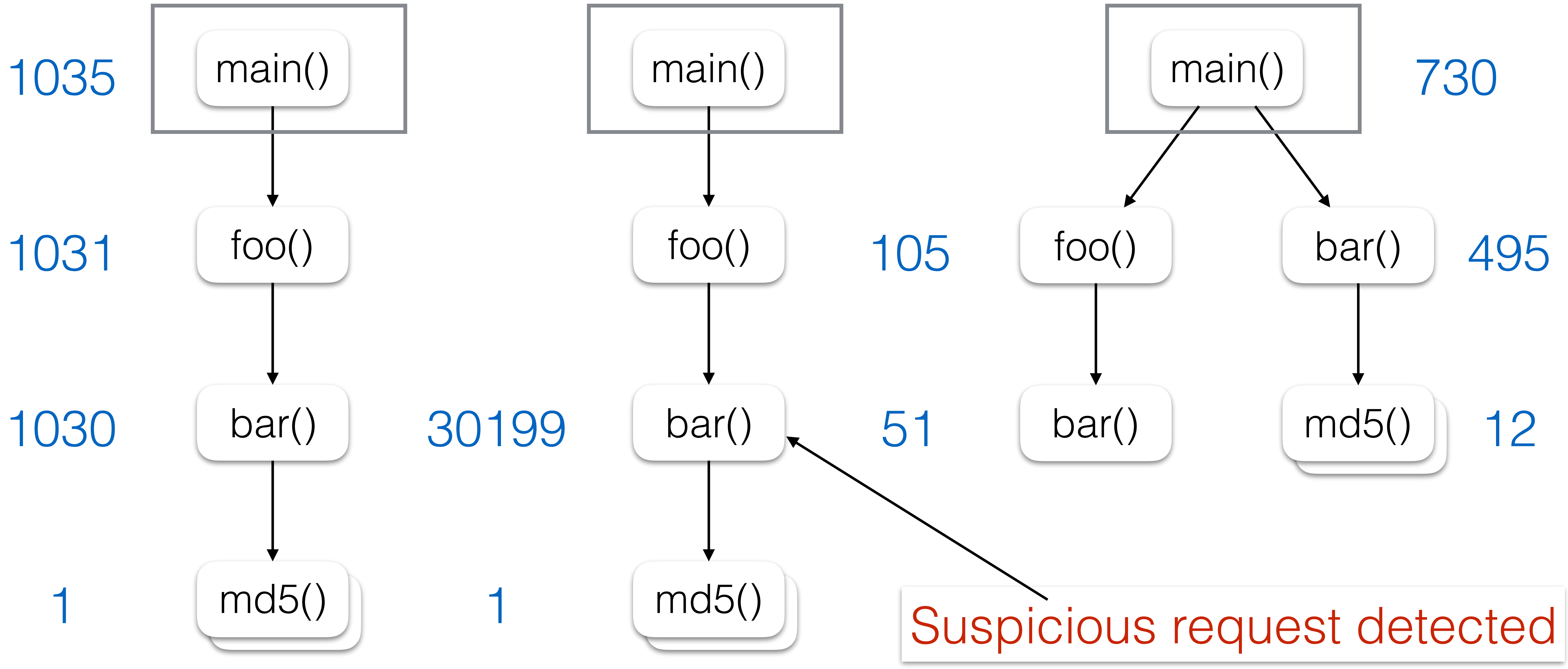


- How to detect the requests causing the DoS?
  - Setting a global timeout?
  - Finding the ones consuming the most CPU time?
- Our approach - finding the ones of which the consumed CPU time is statistically different from their past records
  - Chebyshev's inequality: $P(|X-\mu| > k\sigma) \leq \frac{1}{k^2}$
  - Condition to label suspicious requests: $T_C > min(max(\mu + k \times \sigma, T_{min}), T_{max})$

CPU-Exhaustion DoS Attack Detection (Cont.)

Function Execution Records

Function	CPU time measurements		
a7f2943c	1026	1055	1035
1c39686a	1025	1050	1031
8009ece6	1025	1045	1030
3825111	1	0.95	1 1
61c5ab22	700	750	730
d5d071c9	100	110	105
7589f636	50	45	51
81741924	500	510	495
1f6321a4	10	13	12



Rampart can detect the incident much earlier before bar() returns

Rampart may not determine it as an attack if the CPU usage is low

Probabilisitic Request Termination

- Shall we kill the instances serving the suspicious requests?
  - Not a good idea - false positive requests may deviate not much from the norm
- Our approach - degrading the priority of those requests by
  - Probabilistically terminating the suspicious requests
- A suspicious request would be temporarily suspended or aborted
  - Depending on the current server load and the times it has been suspended

CPU-Exhaustion DoS Attack Blocking

- Is the current design good enough?
  - No, the attackers can still consume the CPU until an alarm
- We need to deploy filters to block follow-up attack requests
  - Requested URI, the request parameters, and the network address
- Are we good to go?
  - A persistent filter - What if it is a false positive filter?
  - A temporary filter - What if the attacker just waits?

The Exploratory Algorithm

- An algorithm to adaptively control the lifetime of a filter
    - Block all matched requests in a primary lifespan
    - Assume we were wrong, *i.e.*, it was a false positive filter
    - Explore the result if it was deactivated continuously until
        - The secondary lifespan expires AND no attack is detected
            - It was a false positive filter OR the attackers had stopped
        - or, an attack is detected again before the expiry of the 2nd lifespan
            - Reset the filter with a longer primary lifespan to penalize the attacker
- The algorithm controls the upper bound of the rate that one attacker can cause CPU-exhaustion DoS

Performance Optimizations

- Avoid unnecessary system calls
  - Disable profiling for built-in functions
- Control the profiling granularity
  - Profile up to *Max_Prof_Depth* function frames in the stack
- Improve write performance & mitigate contention
  - Batch processing measurements with dedicated daemon
- Limit the profiling rate
  - Sampling

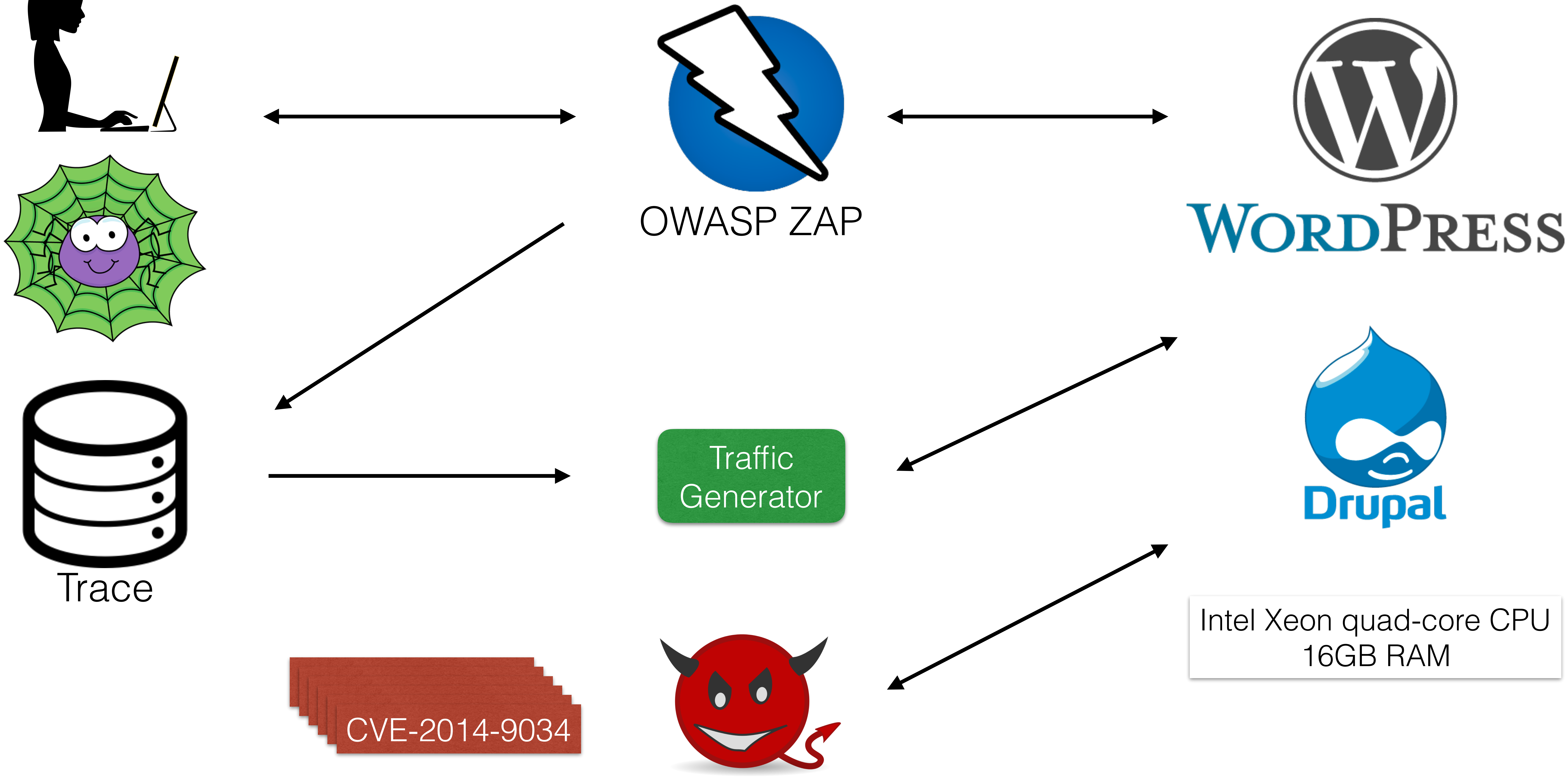
Implementation

- An extension to the PHP Zend engine
  - 2K lines of C code
  - Linux - getrusage() for measuring CPU time
- A separate batch processing daemon
  - 400 lines of Python code
- Why PHP?
  - It is still the most popular server-side programming language

Outline

- Background & Motivation
- Rampart
- Performance Evaluation
- Mitigation Evaluation

Setup


OWASP ZAP
WordPress
Trace
Traffic Generator
Drupal
Intel Xeon quad-core CPU
16GB RAM
CVE-2014-9034

Performance Measurements

Baseline server performance

Application	Benchmark	User Instances 8	16	32	64	96	128
Drupal	ARPT (ms)	277.5	361.8	398.1	502.4	607.3	717.5
	CPU (%)	19.47	24.83	32.21	47.18	59.97	70.53
Wordpress	ARPT (ms)	20.8	21.7	22.5	38.9	85.6	144.7
	CPU (%)	13.47	22.63	42.21	73.03	86.72	90.11

Rampart performance

Application	Benchmark	Max_Prof_Depth 1	3	5	7	9	11
Drupal	ARPT (ms)	397.6	389.0	400.9	393.0	413.6	412.6
	CPU (%)	34.53	34.80	35.62	36.32	38.52	40.94
	# Unique Funcs	12	76	567	1,421	2,473	4,019
	# Funcs	341	2,167	12,677	31,152	53,263	80,186
Wordpress	ARPT (ms)	23.7	23.7	23.5	24.6	29.1	36.4
	CPU (%)	44.25	43.12	49.08	56.56	61.60	69.37
	# Unique Funcs	17	199	846	3,186	7,909	13,337
	# Funcs	422	4,479	15,314	42,957	89,080	136,910

Drupal Overhead
3.41% CPU, 2.8 ms ARPT

Wordpress Overhead
6.87% CPU, 1 ms ARPT

Performance Measurements (Cont.)

CPU usage and request processing time (RPT) over time for 32 users sending requests every 0.1 seconds to Drupal

Outline

- Background & Motivation
- Rampart
- Performance Evaluation
- Mitigation Evaluation

Performance Degradation Caused by Attacks

Application	Benchmark	No Attack	Attack: PHPass [Attackers] 8	Attack: PHPass [Attackers] 16	Attack: XML-RPC [Attackers] 8	Attack: XML-RPC [Attackers] 16
Drupal	ARPT (ms)	398.1	461.2 (1.16x)	519.6 (1.31x)	458.3 (1.15x)	541.7 (1.36x)
	CPU (%)	32.21	88.95	95.05	84.61	94.91
Wordpress	ARPT (ms)	22.5	37.0 (1.64x)	49.0 (2.18x)	31.5 (1.40x)	41.7 (1.86x)
	CPU (%)	42.21	89.71	94.14	83.86	92.08

Effectiveness of Rampart

Application	Benchmark	CPU Threshold for Attack							
		50%				75%			
		PHPass [Attackers]		XML-RPC [Attackers]		PHPass [Attackers]		XML-RPC [Attackers]	
		8	16	8	16	8	16	8	16
Drupal	ARPT-U (ms)	394.7	427.1	423.4	460.4	400.9	418.6	437.4	471.6
	ARPT-A (ms)	203.6	228.3	148.1	172.2	258.9	166.6	160.4	181.0
	CPU (%)	38.51	38.76	36.30	37.68	38.84	39.62	36.30	37.73
	FPR (%)	0.60	0.00	0.25	0.00	0.69	0.00	0.15	0.00
Wordpress	ARPT-U (ms)	24.1	26.1	25.6	26.8	24.4	26.1	24.5	25.1
	ARPT-A (ms)	142.1	234.4	205.9	220.5	152.8	242.3	226.3	180.2
	CPU (%)	45.92	51.40	49.89	50.74	49.15	50.98	50.91	52.14
	FPR (%)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Drupal baseline performance
ARPT-U: 398.1 ms
CPU: 32.21 %

Wordpress baseline performance
ARPT-U: 22.5 ms
CPU: 42.21 %

Performance Degradation Caused by Attacks (Cont.)

CPU usage and RPT over time for 8 PHPass attackers on Drupal without Rampart

Effectiveness of Rampart (Cont.)

CPU usage and RPT over time for 8 PHPass attackers on Drupal with Rampart enabled

Summary

- Rampart performs context-sensitive function-level program profiling to learn function execution models from historical observations
- Rampart detects and mitigates CPU-exhaustion DoS attacks using statistical methods
- Rampart adaptively synthesizes and updates filtering rules to block future attack requests
- Rampart can effectively and efficiently protect web applications from CPU-exhaustion DoS attacks

Thank you!

Q & A