A Prevention Model for Algorithmic Complexity Attacks

Ms Suraiya Khan,

Dr Issa Traore

Information Security and Object Technology (ISOT) Lab
University of Victoria
Victoria, BC, Canada
http://www.isot.ece.uvic.ca

Content

1. Preamble

The SPIDeR Project

Network Anomalies Detectors

- Structure
- Frequency
- Load
- Randomness

Network Forensics

Host Anomalies Detectors

- Masquerade
- Privilege Esc.
- DoS

Intrusion Response

2. Introduction

Context

• DoS: second largest cause of monetary loss according to a survey by FBI/CSI.

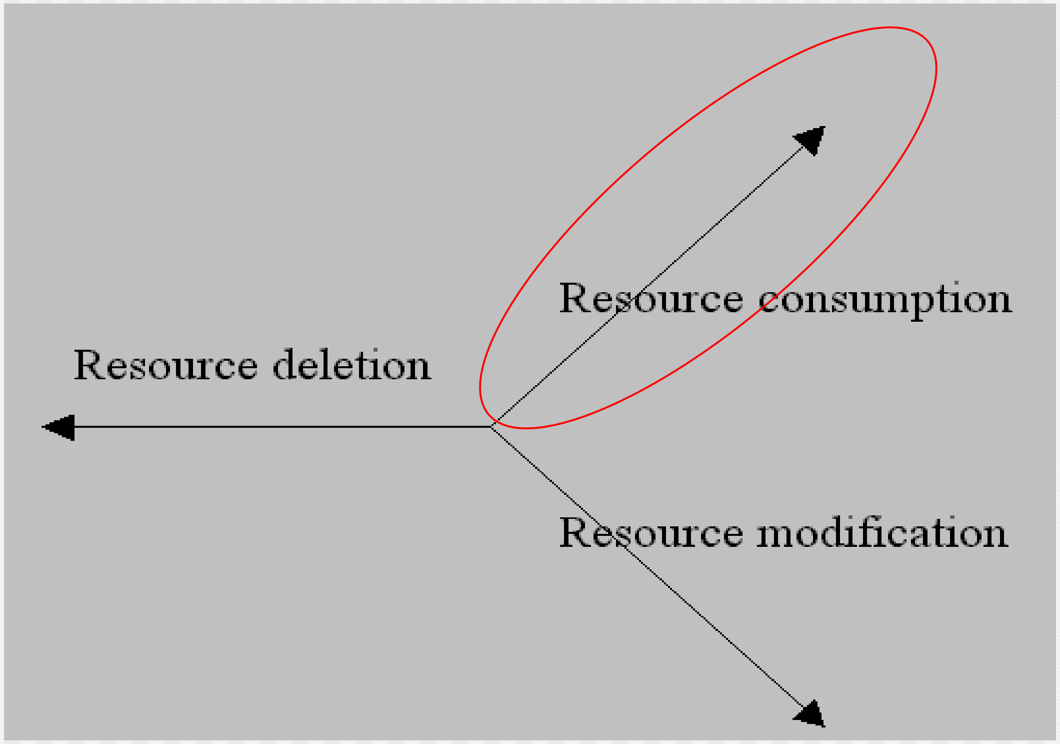
• Over $65M loss in year 2003 is reported by 530 organizations who participated in the survey.

• So necessity to develop DoS protection mechanisms.

- effective DoS protection requires effective DoS detection

2. Introduction (ctd.)

DoS Attack Dimensions



2. Introduction (ctd.)

Resource Consumption Attacks

- **Flooding:** Attack sends too many requests to a system resource.
  - Increases arrival rate.
- **Complexity:** Attack sends many lengthy requests to the resource.
  - Requires more resources for a request than what is typical.
  - May not increase the arrival rate significantly.

2. Introduction (ctd.)

Objectives of Our Work

- Develop prevention mechanisms against complexity attacks.
- Use Service time to detect probable complexity attack requests and drop them.

3. Complexity Attack

Complexity Attack

- Consists of exploiting the working principles of algorithms running on computing systems.
- Made possible when the average case complexity of an algorithm is much lower than the worst-case time or space complexity
- Since deterministic algorithms are the most vulnerable, randomization is used as solution, but:

-this lacks flexibility, and

-it has been shown recently that randomized algorithms are also vulnerable.

3. Complexity Attack (ctd.)

Some Algorithms Prone to Complexity Attacks:

- Quick sort
- Hashing
- Pattern Matching
- Java Byte Code Verification
- B+ Tree

Example of Complexity Attacks

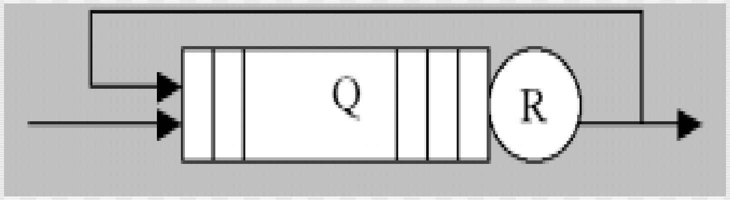
Quick Sort: **used to sort large number of elements.**

Average case: O(*n*log*n*)

Worst case: $O(n^2)$

4. Attack Prevention

Impact of Attacks on Response Time



•Response Time = Waiting Time + Service Time

- ***Waiting time:*** Depends on how many higher priority requests are in the queue.

- ***Service time:*** Time when the request gets service from the resource.

4. Attack Prevention (ctd.)

Possible detection Principles

1. Input size
2. Likelihood of particular service time
3. Temporal density of less likely input (in terms of service time or input size).

Using Service Time

Request Service Time can be analyzed and request can be dropped in two ways:

- *During Actual execution (Delayed)* refer to most likely service time for that input size.
- *Before execution begins:* using input property scanning and service time look-ahead (may have high complexity).

4. Attack Prevention (ctd.)

Prevention model

Compute for each request: ***<ExecutionTime, p_r>***

- ExecutionTime: estimated request execution time (here refers to service time)
- p_r: drop probability in case the request doesn't finish in estimated time

$$ExecutionTime = f(input\_characteristics, object, state, algorithm).$$

Example: Linux 'ls' command:

- Input Characteristics: semantics of arguments and flags.
- Object: directory structure.
- State: present content of the directory structure.
- Algorithm: 'ls' program (necessary to identify which algorithm we are dealing with).

4. Attack Prevention (ctd.)

Example: Quick Sort and delayed drop scheme

$$ExecutionTime = f\left(input\_characteristics, -, -, a\lg orithm\right).$$

- Input Characteristics: Number of elements *n*.
- Object: Don't care.
- State: Don't care.
- Algorithm: glib2.0's ***g_qsort_with_data*** .

Execution Time: computed using regression analysis

4. Attack Prevention (ctd.)

Regression Analysis

For Quick Sort and Delayed drop scheme:

1.Maximum of Most likely Service time (offline Analysis) : Linux "time" command –real trace or randomly generated elements.

-generate inputs randomly for each value of n (n varies from 100-314×10^6; uneven jump)

-for each n take several samples, and from sample execution times, take maximum.

2. Adjusted most likely execution time with 40% increase -conservative most likely estimate.

3.We use a fixed threshold or Regression Equation based on conservative most likely time for different *n* (offline analysis).

4. Attack Prevention (ctd.)

Regression Analysis (ctd.)

Fixed threshold

For number of elements *n* ($\leq$ *70,000*) most likely service time is set to 0.252 second

Otherwise the most probable service time

$$Y = 13.1055 \times \left(\frac{n}{3 \times 10^{6}} \right) - 0.0991 . \qquad (1)$$

Detection principle:

Nonconforming request: Test request has consumed more than the conservative most likely time but did not finish yet– probable attack.

5. Evaluation

Settings

1. Pentium 350 MHz
2. Fedora Core
3. Regression (offline analysis)
4. Already consumed time in Service by a process with id “pid” from /proc/pid/stat (runtime analysis)
5. Testing in the presence of complexity attack on deterministic quick sort (written by us) and **randomized quicksort (glib2.0) [*Attack is still possible].**

5. Evaluation (ctd.)

Randomized Algorithm

Worst case normal input: very unlikely

Attack (Worst case) input: Possible

So, drop the request (with probability one), which does not finish within the estimated time .

5. Evaluation (ctd.)

Randomized Algorithm (ctd.)

Number of elements (n)	Predicted time for normal execution (seconds).	Required actual execution time for attack input (seconds).
100	0.252	0.01
1,000	0.252	0.01
5,000	0.252	0.46
5,600	0.252	0.62
10,000	0.252	1.95
50,000	0.252	61.78
150,000	0.4394	344.02

5. Evaluation (ctd.)

Randomized Algorithm (ctd.)

Detection	False positive	Right detection
Offline	None	All Requests with $n \geq 70,000$
Online	None	Same as above and based on the sampling rate and the scanning speed on /proc/pid/stat for all pid.

5. Evaluation (ctd.)

Deterministic Algorithm:

Worst case normal input: likely.

Attack (Worst case) input: Possible.

So, we cannot always drop requests, which do not finish within the estimated time.

Drop nonconforming requests based on

- Random Drop Probability
- Remaining user token
- Temporal density
- CPU Queue size

5. Evaluation (ctd.)

Deterministic Algorithm (ctd.):

All worst case inputs have same size (40,000); continuous attack.

Policy	**Wrong Drop**	**Right Drop**
Random Drop (drop probability = *p*)	*p*	*p*
Temporal density and Queue size	0.19	0.86
User Token, temporal density, and queue size	0.19	0.77

6. Conclusion

Related Works

Reactive:

Gligor: Maximum Waiting Time (waiting time depends on load).

Spatscheck: Resource accounting (like static threshold)

Gal: Code hardening (no detail available).

Proactive:

Crosby: Randomization (inflexible, approximate result,

attack still possible).

6. Conclusion (ctd.)

Our model of detection followed by drop is a reactive approach – some wrong drops.

Future Work:

- We are working on some proactive approaches to supplement the reactive ones.
- Evaluate detection and drop model on other algorithms prone to Complexity Attacks.