APLIKASI ALGORITMA DEVIDE AND CONQUER UNTUK MENINGKATKAN EFISIENSI PADA CLIENT HONEYSPOTS

Ginanjar Pramadita
NIM 13506014
Teknik Informatika Institut Teknologi Bandung
Jalan Ganesha10, Bandung
2008
e-mail: ginan@students.itb.ac.id

ABSTRAK

Makalah ini membahas bagaimana meningkatkan kecepatan untuk menditkesi dan mengidentifikasi malicious server dalam suatu jaringan internet pada client honeyspot dengan menggunakan algoritma devide and conquer.

Kata kunci: Devide and Conquer, Client Honeyspots, Server.

1. PENDAHULUAN

Internet telah menjadi kebutuhan yang penting masyarakat dunia saat ini. Internet telah menjadi sumber informasi, hiburan, dan menjadi alat komunikasi utama baik di lingkungan rumah atau oun di kantor.

Namun demikian, konektivitas internet menimbulkan ancaman keamanan. *Unauthorized access*, *denail service*, atau penguasaan sepenuhnya terhadap komputer oleh *malicious user* adalah contoh dari ancaman keamanan yang menyerang internet. Karena internet adalah jaringan global, serangan dapat dilakukan dari tempat mana pun di dunia dengan *anonymity* yang tinggi. Oleh karena itu, pada umumnya, komputer yang terhubung dengan internet setidaknya telah memiliki antivirus dan firewall. Tindakan ini terbukti telah cukup sukses untuk menangkal ancaman-ancaman keamanan di internet.

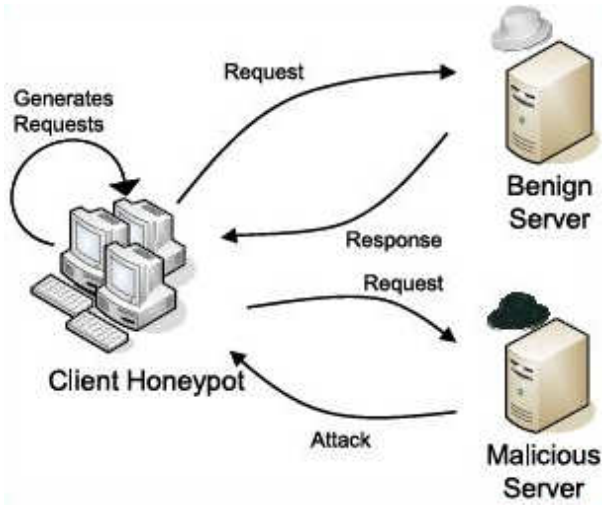
Setelah media penyerangan dihambat oleh aplikasi pertahanan, *malicious user* selanjutnya mencari jalur yang tidak terlindungi untuk menyerang. Serangan ini disebut *client-side attack* yang memiliki target aplikasi klien. Setelah klien mengakses *malicious server*, server selanjutnya mengirimkan serangan kepada pihak klien sebagai bagian dari *request* oleh klien. Contoh yang biasa dari serangan ini adalah server web yang menyerang *web browser*. Setelah *request* dari *web browser* untuk mendapatkan konten, server akan mengirimkan halaman-halaman *malicious* yang akan menyerang *browser*. Jika berhasil, pertahanan tradisional seperti antivirus dan firewall pun akan terinfeksi.

Untuk mengembangkan metoda pertahanan baru terhadap ancaman-ancaman tersebut, dibutuhkan kajian yang lebih mendalam terhadap *malicious server*. Kajian utama terhadap server-server tersebut adalah untuk mencari server-server mana saja yang menjadi *malicous* dalam suatu jaringan, khususnya internet. Untungnya, *malicious server* perlu *accessable* agar penyerangannya berhasil. Peluang inilah yang memungkinkan untuk menemukan *malicious server*. ***Client honeypots*** adalah teknologi baru yang digunakan untuk proses pencarian tersebut. Setelah *malicious server* diidentifikasi, akses terhadapnya dapat dihambat atau peraturan hukum dapat dilibatkan sebagai bantuan untuk menutup element ini dari jaringan.

Namun demikian, *Client honeypots* dihadapkan oleh tantangan-tantangan dari sisi internalnya. Pertama, harus "meraba-raba" internet yang memilik jutaan server. Mencari *malicious server* akan sama halnya dengan mencari jarum di tumpukan jerami. Kecepatan akan menjadi aspek yang sangan krusial untuk mengidentifikasi *malicious server* secara cepat dan melakukan pertahanan terhadapnya. *Client honeypots* yang ada saat ini memiliki kinerja yang lambat. Algoritma-algoritma penditeksi yang berdasarkan pada *monitoring authorized state changes* dari *client honeypots* sangatlah "mahal". Algoritma-algoritma tersebut membutuhkan waktu dan *resource* yang besar. Berdasarkan pada penelitian *client honeypots* yang dikembangkan *Microsoft Research*, kebanyakan *malicious website* harus menginstall terlebih dahulu *mailicious file* antara 30 detik. Berhadapan dengan kuantitas server di internet, durasi 30 detik untuk mengklasifikasi server sebagai *malicious server* membuat pencarian komprehensif di internet menjadi tidak mungkin.

Lalu, bagaimana untuk meningkatkan performansi *client honeypots*? Salah satu caranya adalah dengan menerapkan algoritma *devide and conquer* sebagai jalan *client honeypots* untuk berinteraksi dengan *malicious server* dan mengidentifikasi server tersebut. Topik ini akan dijelaskan pada bagian selanjutnya.

2. CLIENT HONEYPOTS



Gambar 1. Arsitektur Client Honeypots

Setelah *Client honeypots* menemukan server yang diduga *malicious server* dalam suatu jaringan, khususnya internet, selanjutnya mengenerasi antrian permintaan dari server (*server request*). Setelah itu, *client honeypots* mengirimkan *request* tersebut satu per satu ke server dan menerima respon dari server. Setelah menerima respon, *client honeypots* dapat menganalisis dan menentukan apakah suatu server tergolong berbahaya (*malicious server*) atau tidak (*benign server*).

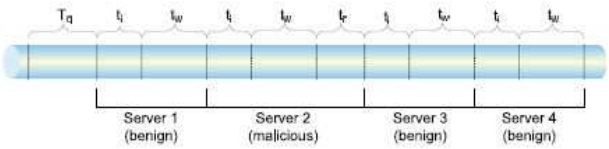
Klasifikasi ini berdasarkan pada monitoring terhadap sistem untuk *unauthorized state changes* yang terjadi pada sistem setelah *client honeypots* berinteraksi dengan server. *Client honeypots* adalah mesin yang berdedikasi dan *unauthorized state changes* seperti proses baru, instalasi file baru, dll, dapat segera dideteksi karena tidak ada aktivitas lain yang terjadi pada *client honeypots*. Sekali *state change* terdeteksi dan klasifikasi telah ditentukan, mesin harus di-*reset* menjadi *state* yang bersih sebelum dapat digunakan kembali untuk berinteraksi dengan server lain.

Penemuan *malicious web server*, misalnya melalui halaman web dari *browser*, akan menyebabkan berbagai perubahan *state* pada sistem. Beberapa perubahan tersebut antara lain, penulisn file ke dalam cache. Ini adalah *authorized event* yang akan diabaikan untuk membuat klasifikasi yang menjadikan suatu web berbahaya atau tidak. Namun, jika ada file *exe* muncul pada folder *start-up*, kita dapat mengklasifikasi halaman web tersebut sebgai *malicious* karena hanya serangan yang terorganisasi dari halaman web yang dapat menyebabkan penempatan file tersebut di folder *start-up*.

Kecepatan diteksi terhadap *mallicious server* dipengaruhi oleh berbagai faktor. Pertama, teknologi yang mendasari akan bagaimana suatu perubahan *state* dideteksi. Sebagai contoh pada *HoneyClient*, salah satu *client honeypots* yang ada saat ini, menggunkan *snapshots* yang memakan waktu lama sedangkan implementasi yang lain menggunakan *event trigers* yang melakukan diteksi tepat pada saat perubahan *state* terjadi. Faktor yang lain adalah:

- waktu total t_t untuk menyelidiki kumpulan n server.
- *Bandwidth* jaringan b
- ukuran rata-rata dari *request*/*response* s, yang mempengatuhi waktu t_i untuk menerima respon server.
- waktu untuk *reset* t_r
- persentasi *malicious server* yang berada di jaringan p_m
- delay terhadap klasifikasi terakhir t_w
- Durasi penerimaan dan analisi terhadap respon server T_q (bersifat konstan)

Setelah antrian *server request* dibuat, setiap server dikunjungi. Setelah selesai kunjungan, *client honeypots* menunggu sebelum memeriksa perubahan *state* pada sistem untuk mengklasifikasi apakah *malicious server* atau *banign server*. Jika server telah diputuskan sebagai *malicious*, *state* pada sistem di-*reset*.



Gambar 2. Contoh Durasi Algoritma Sekuensial

Berikut ini adalah *pseudocode* algortma tersebut.

```
function examine_servers_sequentially()
  server_queue = create_server_queue()
  while(server_queue.next())
    server = server_queue.next()
    visit(server)
    classification = check_state()
    if (classification == MALICIOUS)
      report_state_change()
      reset_state()
    end
  end
end
```

algoritma tersebut memiliki komleksitas yang relatif cukup besar, yaitu $O(n)$ sebagaimana waktu untuk memeriksa server meningkat sejalan dengan dengan konstan C_{Seq} untuk setiap *additional* server yang ditunjukan Gambar 2 di atas. Waktu total untuk memeriksa kumpulan server n adalah sebagai sebagaimana akan diuraikan melalui persmaan berikut:

$$\begin{aligned} t_t &= T_q + C_{Seq} n \\ &= T_q + \left(p_m \left(t_i + t_w + t_r \right) + \left(1 - p_m \right) \left(t_i + t_w \right) \right) n \end{aligned} \quad (1)$$

Dimana t_i = s/b

Dengan kompleksitas yang sebesar $O(n)$ tersebut, tentulah algoritma ini akan sangan tidak efisien, mengingat dalam jaringan internet akan ada jutaan server yang harus diperiksa. Oleh karenanya, kita memerlukan algoritma yang lebih efisien, diantaranya adalah *devide and conquer*.

3. PENGGUNAAN ALGORITMA DEVIDE AND CONQUER

Dengan menggunakan algoritma ini, kompleksitas waktu akan turun dari $O(n)$ menjadi $O(log(n))$.

Setiap server request dibagi ke dalam *buffer of size k* dan memproses server request pada buffer. Perubahan *state* hanya diperiksa setelah semua respon dari server diterima. Jika respon dari *malicious server* terditeksi, buffer akan dibagi menjadi dua dan algoritma akan secara rekursif menerapkan hal yang sama terhadap kedua bagian tersebut. Jika ukuran buffer telah menyusut menjadi 1 dan klasifikasi *malicious* telah dibuat, maka algoritma telah mengidentifikasi respon server yang menimbulkan *malicious.*

Membagi kumpulan server menjadi *groups* yang berukuran *k* akan memilih *malicious server* berdasarkan distribusi binomial. Banyaknya operasi yang dilakukan untuk mengidentifikasi setiap *malicious server* bergantung pada berapa banyak *malicious server* yang telah dipilih (*selected*). Berikut ini adalah *pseudocode* dari algoritma ini.

```
functon visit_and_analize_buffer(buffer)
  while(buffer.next())
    server = buffer.next()
    visit(server)
  end

  wait(INTERVAL)
  classification = check_state()
  if (classificaton == MALICIOUS)
    reset_state()
    if (buffer.size == 1)
      report_state_change()
    else
      first_half = buffer.get_first_half
      visit_and_analyze_buffer(first_half)
      sec_half = buffer.get_sec_half
      visit_and_analyze_buffer(sec_half)
    end
  end
end
```

```
function examine_servers_dac()
  server_queue = create_server_queue()
  while(server_queue.next())
    buffer = get_buffer(server_queue, SIZE)
    cache_buffer(buffer)
    visit_and_analyze_buffer(buffer)
  end
end
```

Jika tidak ada *malicious serve*, algoritma akan beroperasi satu kali pada buffer lalu keluar. Jika ada sebuah *malicious server* pada suatu *group*, algoritma akan beroperasi $2log_2(k) + 1$ kali pada buffer untuk mengidentifikasi *malicious server*. Jika ada lebih dari satu *malicious server* pada suatu *group*, algoritma akan melakukan traversal ke dalam pohon biner untuk setiap respon *m* dari *malicious server* sehingga akan ada $m(2log_2(k) + 1)$ operasi. Berikut ini persamaan matematis algoritma ini.

Untuk kasus terburuk, yaitu untuk mengidentifikasi semua server adalah:

$$op(k,m) = \begin{cases} 1 & \text{jika } m = 0 \\ m(2\log_2(k) + 1) & \\ 2m\log_2(m) - m + 1 & \text{jika } m > 0 \end{cases} \quad (2)$$

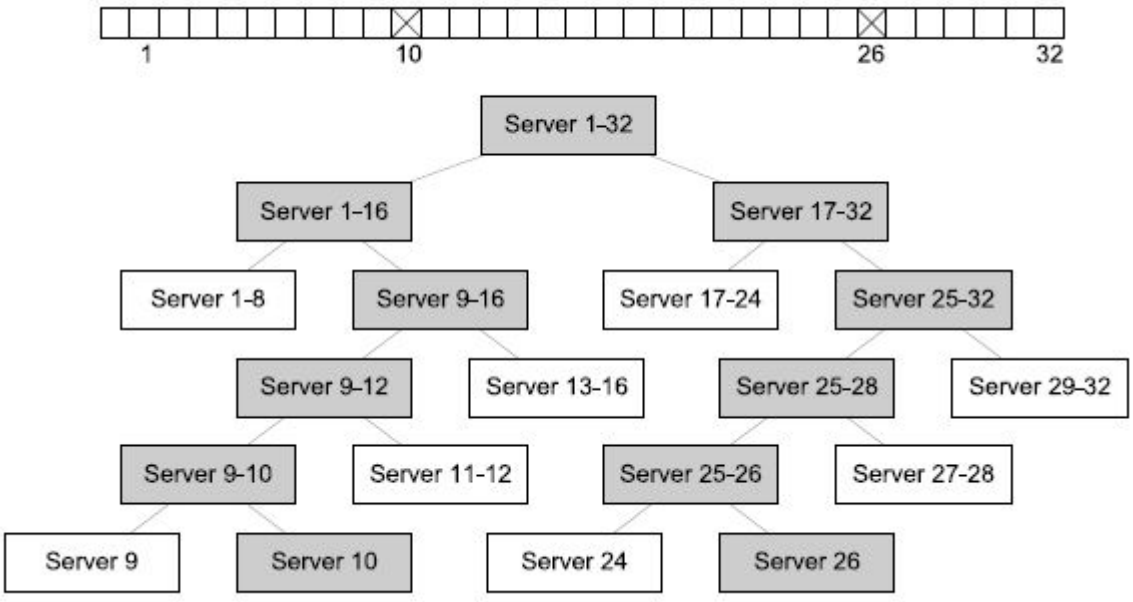
Banyaknya operasi yang dieksekusi yang tidak mengidentifikasi *malicious server* adalah:

$$op(k,m) = \begin{cases} 1 & \text{jika } m = 0 \\ \log_2(k) + 1 & \\ m(\log_2(m) + 1) & \text{jika } m > 0 \end{cases} \quad (3)$$

Sedangkan banyaknya operasi yang dieksekusi untuk mengidentifikasi *malicious server* adalah:

$$op_m(k,m) = m(\log_2(k)\log_2(m) + 2) \quad (4)$$

Gambar 3 di bawah ini menampilkan contoh dari algoritma *devide and conquer*.



Gambar 3. Contoh Algoritma Devide and Conquer

Buffer yang berisi 32 server diperiksa oleh *client honeypots* menggunakan algoritma *devide and conquer* seperti dijeleskan di atas. Pertama, server 1 – 32 diperiksa. Sebuah *malicious server* terditeksi pada buffer, jadi buffer dibagi menjadi dua dan server 1 – 16 serta server 17 – 32 diperiksa. Karena kedua bagian diindikasikan terdapat *malicious server*, maka buffer dibagi kembali dan diperiksa (server 1 – 8, 9 – 16, 17 – 24, dan 25 – 32). Dalam buffer dengan server 1 – 8 dan 17 – 24, tidak ada *malicious server*, jadi tidak ada investigasi lebih lanjut terhadap buffer ini. Pada buffer lainnya, *malicious server* teridentifikasi dan algoritma ini secara rekursif akan memproses hingga *malicious server* 10 dan 26 teridentifikasi. Pohon ini ditraversal dua kali untuk mengidentifikasi setiap server, namun percabangan menempati setelah buffer dengan server 1 – 16 dan 16 – 32 teridentifikasi sebagai *malicious server*. Total terdapat 19 operasi dengan 11 operasi membutuhkan *reset state* pada *client honeyspot.*

Sebagaimana dijelaskan di atas, banyaknya operasi untuk mengidentifikasi *malicious server* dalam suatu buffer ditentukan dari banyaknya *malicious server* dalah buffer tersebut. Banyaknya *malicious server m* yang dipilih dengan ukuran buffer *k* dan persentasinya adalah p_m diuraikan dalam distribusi binomial berikut ini:

$$f(m;k;p_m) = \binom{k}{m} p_m^m (1-p_m)^{k-m} \qquad (5)$$

Ukuran buffer *k* dapat di-set berapa pun berdasarkan persamaan (2). Namun, ada nilai baik dan buruk bagi *k*. Jika nilai *k* terlalu kecil, maka algoritma ini tidak jauh beda dengan algoritma sekuensial. Jika *k* terlalu besar, akan dipilih terlalu banyak *malicious server* dalam buffer sehingga algoritma ini menjadi tidak efisien lagi. Tabel berikut menunjukan nilai optimum bagi *k*.

Tabel 1 Nilai optimum ukuran buffer bergantung pada p_m

p_m	Ukuran optimum buffer *k*
0.005	80
0.010	40
0.015	27
0.020	20
0.025	16
0.030	13
0.035	11
0.040	10
0.045	09
0.050	08

4. KESIMPULAN

Dengan menggunakan algoritma *devide and conquer*, efisiensi meningkat sekitar 72% jika dibanding dengan traversal sekuensial pada *client honeyspots*. Seperti telah dijelaskan di atas, efisiensi pada *client honeyspot* bergantung pada implementasi dan *nature* dari server yang diperiksa.

Kemampuan untuk memriksa kumpulan server lebih cepat dengan algoritma *devide and conquer* dan delay

untuk klasifikasi lebih cepat dengan kemampuan untuk memeriksa dan mengidentifikasi sejumlah server menggunakan *resource* yang sama dan memprediksi serangan yang tertunda oleh *malicious server*.

REFERENSI

[1] Munir, Rinaldi. *Diktat Kuliah IF2251 Strategi Algoritmik*. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. 2006.

[2] http://www.honeyclient.org/trac. waktu akses: 16 Mei 2008 pukul 13.00.

[3] http://www.nz-honeynet.org/capture.html. waktu akses: 19 Mei 2008 pukul 13.30.

[4] Seifert, Christian. "Improve the detaction speed of high client honeyspots"

.