Analisis Aplikasi Pencarian *Exploitable Bugs* Secara Otomatis

Setia Juli Irzal Ismail

Teknik Komputer, Politeknik Telkom
jul@politekniktelkom.ac.id

Abstrak

Aplikasi tidak pernah bisa lepas dari kesalahan (*bugs)*. Tetapi,*bugs* yang dapat menimbulkan celah keamanan dan bisa dieksploitasi *(exploitable bugs)* oleh penyerang merupakan *bugs* yang paling berbahaya. *Exploitable bugs* ini yang menjadi prioritas dan harus ditanggulangi terlebih dahulu. Untuk menentukan apakah *bugs* tersebut bisa dieksploitasi, memerlukan waktu dan pengetahuan teknis tertentu. Untuk itu telah dikembangkan beberapa metode dan aplikasi untuk melakukan pencarian *exploitable bugs* secara otomatis. Pada paper ini dilakukan analisa terhadap beberapa aplikasi yang melakukan pencarian *exploitable bugs* secara otomatis. Aplikasi yang dianalisa adalah SAGE, AEG dan MAYHEM. SAGE mengembangkan metode *whiteboxfuzztesting* yang didasarkan pada pengujian *symbolicexecution* dan *dynamictestgeneration*. Selain melakukan pencarian *exploitable bugs* secara otomatis, AEG juga dapat menghasilkan *exploit* yang dapat digunakan pada *bugs* yang ditemukan. MAYHEM mengembangkan metode *hybrid symbolicexecution* dan *index-based memory modeling* sehingga dapat melakukan pencarian *exploitable bugs* pada *binary code*

Kata kunci: exploitable bugs, *symbolicexecution, index*-based *memory* modeling, hybrid *symbolic* excution, *fuzztesting, dynamictestgeneration*

Abstract

Almost all applicationscontainbugs. However bugs that can be exploited (exploitablebugs) by the attacker is the most dangerous bugs. Exploitable bugs is a priority and must be handled first. To determine whether these bugs can be exploited, requires time and specific technical knowledge. It has been developed some methods and applications for searching exploitable bugs automatically. In this paper we conducted an analysis of several applications which searches exploitable bugsautomatically. These Aplications are SAGE, AEG and MAYHEM. SAGE develop *white-boxfuzztesting* method based on *symbolicexecution* and *dynamictestgeneration*. AEG has ability not just *find*ing exploitable bugs automatically , but itcan also produce an exploit for that specific bugs. MAYHEM developed a hybrid method of*symbolicexecution* and *index*-based modeling *memory* so it can do a search of exploitable bugs in binary *code*

Keywords: exploitable bugs, *symbolicexecution, index*-based *memory* modeling, hybrid *symbolic* excution, *fuzztesting, dynamictestgeneration*

1. Pendahuluan

Aplikasi tidak bisa lepas dari kesalahan (*bugs)*. Terdapat sangat banyak jenis *bugs*. Contohnya saat ini pada *database bugs*, Ubuntu Linux memiliki 102.385 *bugs* yang harus ditangani[1]. Bugs yang paling berbahaya adalah bugs yang dapat menimbulkan celah keamanan serta dapat dieksploitasi oleh penyerang *(exploitable bugs).Exploitable bugs* inilah yang harus menjadi prioritas untuk ditanggulangi (*patched*). Hanya saja untuk menentukan apakah bugs tersebut bisa dieksploitasi atau tidak? diperlukan waktu dan pengetahuan teknis tertentu. Untuk itu dikembangkan beberapa aplikasi dan metode untuk melakukan pencarian *exploitable bugs* secara otomatis.

Berbagai penelitian telah dilakukan untuk mengembangkan metode dan aplikasi pencarian *exploitable bugs* secara otomatis, diantaranya adalah CUTE[2], BitBlaze[3], KLEE[4], SAGE[5], McVeto[6], AEG[7], S2E[8], MAYHEM[9] dan lainnya. Pengembangan aplikasi tersebut untuk melakukan pencarian *exploitable bugs* didasarkan dengan melakukan pengujian *symbolicexecution*[10]. Tetapi masing-masing aplikasi kemudian menambahkan metode yang berbeda untuk melengkapi pengujian *symbolicexecution* tersebut. Pada paper ini akan dilakukan analisa terhadap tiga aplikasi, yaitu SAGE, AEG dan MAYHEM. Pemilihan ketiga paper ini untuk mewakili beberapa metode yang berbeda pada pencarian exploitable bugs.

SAGE (*Scalabale, Automated, Guided Execution*) merupakan sebuah metode *whiteboxfuzztesting* alternatif yang dikembangkan dari pengujian *symbolicexecution* dan *dynamictestgeneration*. Teknik ini merupakan modifikasi dari *Fuzztesting* yang umumnya menggunakan pengujian *blackbox* dan *static analysis*. *Fuzztesting* dianggap sebagai cara yang efektif untuk menemukan celah keamanan pada Aplikasi. *Fuzztestingtools* biasanya memasukkan

input secara *random* pada aplikasi dan menguji hasil outputnya. Sementara SAGE melakukan analisa terhadap hasil pengujian dari sebuah *input*, kemudian algoritma menghasilkan *input* baru yang menjalankan *path* yang berbeda pada *program*. Sehingga tidak terjadi perulangan *path* pengujian. Dengan pendekatan ini *bug* dapat ditemukan lebih cepat dan . Dari hasil pengujian, SAGE berhasil menemukan celah keamanan MS07-017 ANI yang tidak dapat ditemukan oleh *toolsblackboxfuzzing* dan *static analysis*. SAGE juga mampu menemukan 30 *bugs* baru pada sejumlah aplikasi baru Windows, diantaranya aplikasi *image prosessor,media player* dan *file* dekoder. *Bugs* yang ditemukan dapat digunakan untuk exploitasi pelanggaran akses memori.

Pada sebuah aplikasi yang akan diuji, AEG dapat secara otomatis mencari *bugs* yang dapat dieksploitasi kemudian menghasilkan eksploit yang dapat digunakan terhadap *bugs* yang ditemukan. AEG telah digunakan untuk menguji 14 aplikasi *open source*dan berhasil membuat 16 eksploit *control flow hijacking*. Dua diantaranya merupakan *zero-day exploits* pada celah keamanan yang belum diketahui. AEG juga mengembangkan teknik *preconditioned symbolicexecution* pada pencarian *bugs*. Pencarian *bugs* yang dilakukan oleh AEG dilakukan pada tingkat *sourcecode*.

MAYHEM dapat melakukan pencarian *exploitable bugs* secara otomatis pada *binary code*, sehingga tidak diperlukan *debugging* pada aplikasi yang akan diuji. MAYHEM juga mengembangkan metode pencarian *hybrid symbolicexecution* yang menggabungkan teknik *offline* dan *online symbolicexecution*. Selain itu digunakan juga teknik *index-based memory modeling* yang memudahkan pencarian *bugs* pada level *binary*. Pengujian aplikasi pada Linux dan Windows dengan MAYHEM telah dilakukan dan berhasil menemukan 29 celah keamanan yang dapat dieksploitasi, 2 diantaranya merupakan celah keamanan baru.

2. SAGE

FuzzTesting telah dianggap sebagai cara yang cepat dan ekonomis untuk menemukan celah keamanan pada aplikasi. *FuzzTestingtools* biasanya memasukkan *input* yang dihasilkan secara *random* kepada aplikasi yang diuji dan kemudian menganalisa *output* dari aplikasi tersebut. Walaupun begitu *FuzzTesting* yang merupakan pengujian *blackbox* memiliki banyak kelemahan. *Randomtesting* ini umumnya tidak mampu menguji *code* secara keseluruhan (*lowcodecoverage*), karena *input* dihasilkan secara *random*, sehingga bisa saja tidak semua *path* pada *code* teruji. Dari sisi keamanan kelemahan ini sangat berbahaya. Contohnya pengujian dengan *fuzztesting* bisa saja tidak menemukan celah keamanan yang sangat berbahaya seperti *bufferoverflow*, karena bagian *code* yang memiliki celah keamanan ini tidak diuji oleh *Fuzztesting*.

Untuk mengatasi kelemahan *Fuzztesting* dapat digunakan pengujian *symbolicexecution* dan *dynamictestgeneration*. Pada metode *dynamictestgeneration*, *input* tidak dihasilkan secara *random*, melainkan merupakan hasil analisa terhadap *outputprogram* dari *input* yang telah ada. Dengan metode ini *input* yang baru akan mengikuti *executionpath* yang berbeda daripada *path* yang dihasilkan oleh *input* yang telah ada. Sehingga tidak terjadi pengulangan *executionpath* pada *program* dan semua *path* yang ada dapat diuji. Hanya saja untuk melakukan pengujian pada semua *path* yang ada, akan membutuhkan *test* vektor yang banyak dan menghabiskan memori. Pengujian ini tidak efektif untuk menguji aplikasi dengan *sourcecode* yang besar.

Untuk itu SAGE mengajukan algoritma baru yang dinamakan *alternativewhite-boxtesting*. Algoritma ini merupakan pengembangan dari pengujian *symbolicexecution* dengan menggunakan teknik optimasi, sehingga dapat mengurangi *test* vector yang diuji. Algoritma ini akan bermanfaat untuk melakukan pengujian pada aplikasi besar yang memiliki *inputfile* yang banyak dan memiliki *executiontrace* yang panjang.

a. *DynamicTestGeneration*

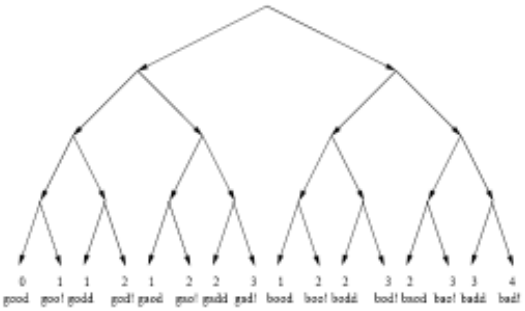
Misalnya kita melakukan pengujian aplikasi pada Gambar 1. Pengujian dengan *fuzz testing* akan sulit menemukan *error*. Karena dengan nilai *input random* yang dihasilkan oleh *fuzz testing tools* akan sulit untuk menemukan semua *execution path* yang mungkin ada. Sementara dengan menggunakan metode *whitebox dynamic test generation*, kesalahan pada aplikasi tersebut akan mudah ditemukan.

```
void top(char input[4]) {
int cnt=0;
if (input[0] == 'b') cnt++;
if (input[1] == 'a') cnt++;
if (input[2] == 'd') cnt++;
if (input[3] == '!') cnt++;
if (cnt >= 3) abort(); // error
}
```

Gambar 1. Contoh Aplikasi [11]

Metode ini akan memulai pengujian dengan memasukkan *input* awal, kemudian melakukan *dynamicsymbolicexecution* untuk menganalisa *path* pada *code* yang dilewati oleh *input* tersebut. Hasil analisa *path* yang dilewati akan menghasilkan *input* baru yang akan melewati *executionpath* yang berbeda. Proses ini akan terus diulangi sampai semua *path* pada *code* telah diuji. Contohnya dengan menggunakan metode ini untuk menguji aplikasi pada Gambar 1 ditemukan 16

executionpath yang ada. Skemanya seperti pada Gambar 2.



Gambar 2 *Executionpath* dari Aplikasi 1 [11]

Metode *dynamicpathgeneration* dapat menemukan semua *executionpath* dari aplikasi yang kita uji, sehingga dapat menguji semua bagian *code* dari aplikasi. Walaupun begitu kelemahan dari metode ini membutuhkan waktu dan memori yang besar.

b. Generational Search

Untuk mengatasi kelemahan dari metode *dynamictestgeneration*, maka SAGE menggunakan algoritma pencarian baru yang dinamakan *Generationalsearch*. Algoritma ini mengurangi *executionpath* yang dicari, menjadi hanya sebagian saja. Walaupun begitu algoritma ini tetap mampu menguji *code*, sehingga *codecoverage* yang diuji cukup besar. Algoritma ini dapat melakukan eksplorasi *path* pada aplikasi besar yang memiliki *input* besar, ribuan variabel *symbolic* dan dengan *path* yang dalam yaitu ribuan juta instruksi. Optimasi jumlah *test* yang dihasilkan dari setiap *symbolicexecution* juga dilakukan, sementara pengulangan langkah pencarian dapat dihindarkan. Dengan algoritma ini, pencarian *bugs* akan lebih cepat dan menyeluruh karena menggunakan metode *heuristik*. Bila terjadi divergensi, algoritma juga dapat melakukan *recovery* dan melanjutkan pencarian.

Generationalsearch ini dibagi menjadi 2 bagian yaitu : algoritma *Search* pada gambar 3 dan perhitungan *newchildren* (*path* baru) pada gambar 4. Algoritma *Search*, digunakan untuk menghasilkan *input* awal (*inputSeed*) pada *worklist* (baris 3), kemudian menjalankan aplikasi untuk menemukan *bugs* pada eksekusi pertama (baris 4). *Input* pada *worklist* kemudian diproses (baris 5) dengan memilih sebuah elemen (baris 6) dan mengembangkannya (baris 7) untuk menghasilkan *input* baru dengan fungsi *ExpandExecution* pada gambar 4. Pada setiap *child*, *program* yang diuji akan menjalankan *input* tersebut. Eksekusi ini akan menguji *error* (baris 10) dan kemudian diberi *Score* (nilai).

```
1  Search(inputSeed){
2    inputSeed.bound = 0;
3    workList = {inputSeed};
4    Run&Check(inputSeed);
5    while (workList not empty) {//new children
6      input = PickFirstItem(workList);
7      childInputs = ExpandExecution(input);
8      while (childInputs not empty) {
9        newInput = PickOneItem(childInputs);
10       Run&Check(newInput);
11       Score(newInput);
12       workList = workList + newInput;
13     }
14   }
15 }
```

Gambar 3 Algoritma Search [11]

Pada Gambar 4 ditunjukkan cara pencarian *path* baru (*children*). Dengan sebuah *input* (baris 1), fungsi *ExpandExecution* akan menjalankan *program* secara *symbolic* dan menghasilkan sebuah *pathconstraint* PC (baris 4). Kemudian algoritma akan melakukan *expand* semua *constraint* pada *pathconstraint*. Jadi dengan algoritma ini semua PC *constraint* akan di-*expand*, tidak hanya pencarian terakhir ataupun yang pertama saja.

```
1  ExpandExecution(input) {
2    childInputs = {};
3    // symbolically execute (program,input)
4    PC = ComputePathConstraint(input);
5    for (j=input.bound; j < |PC|; j++) {
6      if((PC[0..(j-1)] and not(PC[j]))
                    has a solution I){
7        newInput = input + I;
8        newInput.bound = j;
9        childInputs = childInputs + newInput;
10   }
11   return childInputs;
12 }
```

Gambar 4 Menghitung *path* baru[11]

Algoritma *newchildren* ini digunakan untuk menghitung *path* baru dan mencegah terjadinya pengulangan *path*. Untuk mencegah terjadinya perulangan maka digunakan parameter *bound* untuk meembatasi proses backtracking dari *branch* yang telah ada. Dengan Algoritma *generationSearch* ini maka jumlah *input* yang diujikan akan maksimal. Selain itu penggunaan metode *heuristic* dengan sistem prioritas akan menghasilkan pencarian *bugs* yang lebih cepat.

2.3 Sistem SAGE

Algoritma *generationalsearch* ini diimplementasikan pada SAGE (*Scalable, Automated, Guided Execution)*. SAGE dapat menguji *program* pembaca *file* yang berjalan pada Windows dengan membaca *bytes* yang dibaca dari *file* sebagai *input symbolic*. Pencarian *bugs* juga dilakukan pada level *binary* pada x86.

SAGE menjalankan 4 *task* yang berbeda, *Testertask* yang menjalankan fungsi *Run&Check* dengan menjalankan *program* yang diuji dengan sebuah *input* dan menganalisa keluaran yang tidak normal, seperti *access violation exceptions* dan

penggunaan memori yang ekstrem. Apabila *Tester* mendeteksi *error*, maka dia akan menyimpan hasil pengujian dan melakukan *seed* otomatis. *Tracer task* menjalankan aplikasi target dengan *input* yang sama, kemudian menyimpan log yang dapat digunakan oleh *task* berikutnya untuk melakukan reka ulang eksekusi *program* secara *offline*. *Task Coverage Collector* melakukan reka ulang eksekusi. SAGE menggunakan informasi ini pada implementasi fungsi *Score*. Terakhir ada *Symbolic Executor task* yang menjalankan fungsi *Expand Execution* dengan melakukan reka ulang eksekusi yang sudah disimpan sekali lagi. Kali ini untuk mengumpulkan *inputconstraint* dan menghasilkan *input* baru dengan menggunakan *constraintsolver*.

Constraintgeneration pada SAGE berbeda dengan implementasi pada *dynamic test generation* pada dua hal: 1) SAGE menggunakan pendekatan *machine-code-based*. Sementara pada *dynamic test generation* digunakan *source-based instrumentation*. 2) Perbedaan yang kedua, SAGE tidak menggunakan *online instrumentation* seperti pada *dynamic test generation*, melainkan menggunakan *offline trace based constraint generation*.

SAGE menggunakan beberapa teknik optimasi *Constraint*. Optimasi ini dimaksudkan untuk meningkatkan kecepatan *constraint generation* serta mengoptimalkan pemakaian memori. Teknik yang digunakan adalah *tag caching, unrelated constraint elimination, local constraint caching, flip count limit* serta *concretization*.

2.4 Pengujian

Pada bulan Desember 2006 ditemukan celah keamanan pada Microsoft yaitu *bugs* pada proses *parsingfile* dengan format ANI. Metode *blackbox FuzzTesting* tidak berhasil menemukan *bug* ini. Pengujian dengan SAGE telah berhasil menemukan *bugs* ini. Kemudian *bugs* ini ditanggulangi Microsoft dengan dikeluarkannya patch MS07-017. Pengujian berikutnya juga berhasil menemukan celah keamanan pada *file* yang dikompress. Pengujian lainnya juga berhasil menemukan *bugs* pada aplikasi Microsoft Office 2007, pada *parsing* media *file* dan pada *parsing* gambar. *Bugs* ini tidak ditemukan dengan metode *FuzzTesting*.

3 AEG

Automatic ExploitGeneration (AEG) merupakan aplikasi yang dapat mencari celah keamanan pada sebuah sistem, dan secara otomatis membuat eksploit untuk memanfaatkan celah yang ditemukan tersebut. Pencarian *bugs* ini dilakukan pada level *sourcecode*. Metode yang digunakan merupakan pengembangan dari pengujian *symbolicexecution*. Pada AEG dikembangkan metode *preconditionedsymbolicexecution*, dan *pathprioritization* yang dapat memilih *executionpath* mana yang mungkin memiliki *exploit*able *bugs*. Dengan metode ini maka pencarian tidak harus melewati semua *path* yang ada.

AEG juga dapat digunakan untuk meningkatkan pertahanan sistem. Misalnya dengan melakukan *generate* eksploit dan menjalankannya pada *signature* IDS, sehinga IDS dapat mempelajari *signature* serangan baru tanpa harus mendapatkan serangan yang sebenarnya.

Beberapa permasalahan yang mendorong pengembangan AEG adalah tidak efektifnya pencarian *bugs* dengan analisa *sourcecode*, sementara analisis pada tingkat *binary* tidak dapat diukur. AEG melakukan kombinasi analisis pada tingkat *sourcecode* dan tingkat *binary* untuk membuat eksploit pada aplikasi. Permasalahan berikutnya adalah mencari *path* mana yang dapat dieksploitasi diantara ribuan *path* yang ada. Dengan teknik *preconditionedSymbolicexecution*, maka AEG dapat menentukan *path* mana yang dapat dieksploitasi. Kemudian teknik *pathprioritization* yang menggunakan metode *heuristic* dapat menentukan *path* mana yang lebih mungkin untuk dieksploitasi. AEG menghasilkan sebuah sistem *commandline* yang mampu menganalisa *sourcecode* aplikasi, menghasilkan formula *symbolicexecution*, menyelesaikan formula tersebut, melakukan analisis *binary*, menghasilkan *binary*-level *runtimeconstraint*s dan membuat eksploit sebagai *output* yang dapat dijalankan pada sistem yang diuji. Eksploit yang dibuat dapat digunakan pada lingkungan yang berbeda, tapi tidak dirancang untuk dapat menghadapi mekanisme pertahanan seperti address space *randomization* atau windows DEP.

3.1 Komponen AEG

AEG memiliki 6 bagian, yaitu *Pre-PROCESS, SEC-ANALYSIS, BUG FIND, DBA, EXPLOIT GEN dan VERIFY*.

a. *PRE PROCESS*: src $\rightarrow$(B_{gcc}, B_{llvm})
   sourcecode (src) akan dikompile dan diduplikasi, satu berbentuk *binary* B_{gcc} akan digunakan untuk menghasilkan eksploit. Sementara satu lagi berbentuk LLVM byte B_{llvm}akan digunakan pada proses pencarian *bug*. Pengujian memerlukan aplikasi yang akan diuji dalam bentuk *binary* dan *sourcecode*.
b. *SRC-ANALYSIS*: B_{llvm} $\rightarrow$ *max*.
   Komponen ini akan menganalisa *sourcecode* untuk menentukan ukuran maksimum *symbolic* data *max* yang akan dijalankan pada aplikasi. AEG menentukan *max* dengan mencari *bufferstatic* terbesar yang dialokasikan oleh aplikasi. Dengan metode *heuristic* didapatkan *max* setidaknya memiliki ukuran 10% lebih besar dari ukuran *buffer* terbesar.

c. *BUG-FIND* (B_{llvm}, φ, *max*) → (Π_{bug}, *V)*
BUG-FIND akan melakukan analisa *code* LLVM B_{llvm} dan sebuah *safety property* φ, sehingga menghasilkan *output* sebuah *tuple* (Π_{bug}, *V)* untuk setiap celah keamanan yang ditemukan..*BUG-FIND* mencari *bugs* dengan teknik *preconditionedsymbolicexecution* dan menggunakan teknik *heuristicpathprioritization* untuk menentukan *path* mana yang harus diuji terlebih dahulu.

d. DBA (*DynamicBinary* Analysis) : (B_{gcc}, (Π_{bug}, *V*)) → R
Disini dilakukan analisis *binary* secara dinamik pada target *binary* B_{gcc} dengan sebuah *input* dan *runtime* information R. *Input* dihasilkan dengan menyelesaikan *pathconstraint*s $\Pi_{bug.}$ DBA juga menguji *binary* untuk menghasilkan *low*-level *runtime* information (R) seperti celah alamat *buffer* pada *stack*, alamat dari *returnfunction* yang *vulnerable*, dan isi memori *stack* sebelum celah keamanan tersebut dipicu. DBA harus memastikan bahwa semua data yang dikumpulkan akurat.

e. *EXPLOIT* GEN : (Π_{bug}, R) → Π_{bug} ^ $\Pi_{exploit}$
EXPLOIT-GEN menerima *tuple* dengan predikat *path* dari *bug* Π_{bug} dan *runtime* information (R) , kemudian membuat formula untuk sebuah eksploit *control flowhijacking*.

f. VERIFY : (B_{gcc}, Π_{bug} ^ $\Pi_{exploit}$)→ {ε , ⊥ }
VERIFY menerima *input* target *binary* dan sebuah formula eksploit Π_{bug} ^ $\Pi_{exploit}$, dan mengeluarkan *output* eksploit ε. Merupakan bagian *constraintsolver* yang akan melakukan verifikasi apakah *bug* beserta *exploit* yang dibuat telah sesuai.

Algoritma 1 pada Gambar 5 merupakan algoritma yang digunakan untuk menghasilkan eksploit pada AEG.

Algorithm 1: Our AEG exploit generation algorithm

input : *src*: the program's source code
output: {ε, ⊥}: a working exploit or ⊥

```
1 (B_gcc, B_llvm) = Pre-Process(src);
2 max = Src-Analysis(B_llvm);
3 while (Π_bug, V) = Bug-Find(B_llvm, φ, max) do
4 |   R = DBA(B_gcc, (Π_bug, V));
5 |   Π_bug ∧ Π_exploit = Exploit-Gen(Π_bug, R);
6 |   ε = Verify(B_gcc, Π_bug ∧ Π_exploit);
7 |   if ε ≠ ⊥ then
8 |   |   return ε;
9 return ⊥;
```

Gambar 5 Exploit *Generation* Algorithm[12]

3.1 Implementasi

AEG dikembangkan dengan bahasa C++ dan *Phyton*. Komponen *BUG-FIND* yang merupakan *symbolicexecution* dibangun berbasis KLEE[13], dengan tambahan 5000 *line* untuk implementasi teknik precondition *symbolicexecution* dan *heuristicpathprioritization*. DBA dikembangkan dalam bahasa *Phyton*. sementara *Constraintsolver* (Verify) dikembangkan menggunakan STP.

AEG menghasilkan dua tipe eksploit yaitu *Return-to stack eksploit* dan *return to libe eksploit*. Keduanya umum digunakan pada serangan *controlhijacking*. Contoh algoritma *StackOverflow* Return to *Stack* bisa dilihat pada gambar 6

Algorithm 2: Stack-Overflow Return-to-Stack Exploit Predicate Generation Algorithm

input : (bufaddr, &retaddr, μ) = R
output: $\Pi_{exploit}$

```
1 for i = 1 to len(μ) do
2 |   exp_str[i] ← μ[i];          // stack restoration
3 offset ← &retaddr - bufaddr;
4 jmp_target ← offset + 8;   // old ebp + retaddr = 8
5 exp_str[offset] ← jmp_target;          // eip hijack
6 for i = 1 to len(shellcode) do
7 |   exp_str[offset + i] ← shellcode[i];
8 return (Mem[bufaddr] == exp_str[1]) ∧ ... ∧
  (Mem[bufaddr + len(μ) − 1] == exp_str[len(μ)]);
  // Π_exploit
```

Gambar 6 *StackOverflow* Return to *stack*[12]

Return to *stackexploit* memerlukan *shellcode*. AEG memiliki database *shellcode* dengan 2 kelas: *shellcode* standar untuk eksploit lokal dan *binding* serta *reverse binding shellcode* untuk *remoteexploit*. Untuk penempatan *shellcode*, AEG menggunakan pendekatan *Brute-force*. Database *shellcode* yang dimiliki AEG ada sekitar 20 *shellcode* yang berbeda termasuk standar *shellcode* dan alphanumerik *shellcode*. Untuk meningkatkan kehandalan *exploit*, AEG dapat menyesuaikan *offset* pada variabel yang berbeda-beda, selain itu AEG memiliki juga NOP-*sled*. Secara umum NOP-*sled* yang besar akan membuat *exploit* lebih handal, terutama menghadapi perlindungan ASLR. Pada sisi lain NOP-*sled* akan membuat ukuran *payload* menjadi lebih besar, yang menyebabkan eksploit menjadi lebih susah diimplementasikan. AEG memberikan pilihan untuk menghidupkan atau mematikan NOP-*sled* pada *commandline*.

Pengujian terhadap AEG dilakukan terhadap 14 *opensourceprogram*. Pemilihan 14 aplikasi ini diambil dari database *Common Vulnerabilites and Exposures (CVE*), *OpenSourceVulnerability Database (OSVDB*) dan *Exploit*-DB (EDB). Hasil pengujian AEG menemukan celah keamanan yang telah ada, dan menemukan 2 celah keamanan baru yaitu expect-5.43 dan htget-0.93. AEG juga berhasil membuat eksploit dari celah keamanan tersebut. Waktu pengujian sangat cepat yaitu rata-rata adalah 114,6 detik.

Pengujian diatas dilakukan pada mesin Intel Core 2 Duo CPU dengan RAM 4GB dan cache 4 MB L2. Operating sistem yang digunakan adalah Debian Linux 2.6.26-2. *Sourcecodeprogram* dicompile dengan LLVM GCC 2.7 dan GCC 4.2.4 untuk membangun aplikasi. Aplikasi yang diuji merupakan aplikasi asli tanpa modifikasi dan dapat

diunduh secara bebas di internet. Walaupun AEG dibangun berbasis KLEE, tapi dari pengujian yang dilakukan, KLEE hanya mampu mendeteksi *bug*pada aplikasi iwconfig, sementara *bug* pada aplikasi lain tidak berhasil ditemukan.

TABEL 1
JUMLAH EKSPLOIT YANG DIHASILKAN[12]

Program	# of exploits
`iwconfig`	3265
`ncompress`	576
`aeon`	612
`htget`	939
`glftpd`	2201

Pada Tabel 1 terlihat jumlah eksploit yang dihasilkan AEG pada 5 aplikasi yang diuji selama satu jam.

4 MAYHEM

MAYHEM[14] merupakan pengembangan dari AEG, aplikasi ini dapat mencari *exploitablebugs* pada *programbinary*. Semua *bugs* yang ditemukan dapat diuji dengan eksploit yang dihasilkan secara otomatis. Eksploit yang dibuat ini dapat memastikan apakah *bugs* yang ditemukan memiliki celah keamanan. MAYHEM melakukan pencarian pada tingkat *binary* tanpa perlu melakukan *debugging* terlebih dahulu. MAYHEM mencoba mencari teknik untuk melakukan *pathexecution* tanpa menghabiskan memori, kemudian menggunakan metode *symbolicmemoryindex* dimana sebuah alamat penyimpanan tergantung pada *input* user.

Dua teknik yang diperkenalkan oleh MAYHEM adalah : 1) *Hybridsymbolicexecution* yang melakukan kombinasi *online* dan *offline* (*concolic*) *execution* untuk mendapatkan keuntungan dari kedua teknik tersebut, dan 2) pemodelan memori berbasis *index*, sebuah teknik yang secara efisien mampu menjalankan *symbolicmemory* pada level *binary*. Metode deteksi yang digunakan MAYHEM untuk melakukan deteksi *bugs* dan menghasilkan *exploit* didasarkan pada prinsip yang digunakanan oleh AEG.

Pencarian *bugs* yang ada dapat dibagi menjadi dua kategori, 1) *offlineexecutor* yang menjalankan sebuah *singleexecutionpath* dan kemudian dijalankan secara *symbolic*. Metode ini disebut juga *trace-based* atau *concolic executor*, contohnya SAGE[11]. 2) *onlineexecutor* yang menjalankan semua *path* yang ada dengan *single* run , contohnya S2E[15]. *Offlineexecutortools* akan mencari *path* yang baru secara *iterative*. Hal ini akan menghasilkan setiap pengujian bersifat independen dari hasil pengujian yang lainnya. Hanya saja ada kendala pada proses re-*execution* dari *exertion trace*, selain itu mungkin saja terjadi perulangan *path* yang sudah diuji. *Onlinesymbolicexecution* melakukan *trace* pada setiap percabangan. Metode ini menjamin tidak akan terjadi perulangan *path*. Hanya saja memori yang dibutuhkan sangat besar dan akan semakin lambat bila jumlah percabangan semakin banyak. Contoh aplikasi metode ini adalah KLEE[13].

MAYHEM mengkombinasi keunggulan dari dua metode diatas dengan mengusulkan metode *hybridsymbolicexecution*. Metode ini menggunakan metode *online* dan *offlineexecution* secara bergantian. Disini *Hybrid* berfungsi seperti manajer memori pada Operating Sistem. Ketika memori kritis, *hybridengine* mematikan sebuah *executor* yang sedang berjalan dan menyimpan *currentexecutionstate* dan *path* formula. *Thread* dikembalikan dengan mengembalikan formula, menjalankan *program* sampai kembali pada kondisi sebelumnya. MAYHEM mampu menyimpan *path* formula mencegah eksekusi ulang dari intruksi, yang merupakan titik lemah dari *offlineexecution*, dan mampu menggunakan memori jauh lebih efisien daripada *onlineexecution*. Ada tiga kontribusi yang diberikan MAYHEM, yaitu :

1. *HybridExecution*. Skema baru yang membuat kita dapat menyeimbangkan antara kecepatan eksekusi dan penggunaan memori. Metode ini juga memungkinkan MAYHEM untuk mengeksplorasi *path* secara lebih cepat daripada metode yang telah ada.
2. *Index-basedmemory* modeling. Metode ini digunakan sebagai pendekatan untuk melakukan *indexsymbolic* pada level *binary*.
3. *Binary*-only *exploitgeneration*. MAYHEM mampu melakukan pencarian *bug* dan menghasilkan *exploit* secara otomatis tanpa membutuhkan *sourcecode*.

4.1 Arsitektur MAYHEM

MAYHEM terdiri dari 2 proses yang berjalan secara bersama, yaitu *Concrete Executor Client* (CEC) dan sebuah *symbolicExecutor* Sercer (SES). Pada *high-level*, CEC bekerja pada sistem target, SES berjalan pada platform apa saja dan menunggu koneksi dari CEC. CEC mengambil sebuah *programbinary* sebagai *input*, dan melakukan komunikasi dengan SES. SES kemudian melakukan *symbolicexecution* terhadap blok yang dikirim SES, dan menghasilkan *output* beberapa tipe *test* case, termasuk normal *test* case, crash *test* case dan *exploit*s. Langkah yang dilakukan MAYHEM untuk mencari *bugs* dan menghasilkan *exploit* adalah sebagai berikut :

1. Perintah –sym-net 80 400 meminta MAYHEM untuk melakukan *symbolicexecution* pada data yang dibaca pada port 80. MAYHEM dapat mengolah *input* dari files, *environment variables* dan dari jaringan.
2. CEC menjalankan aplikasi yang diuji dan berkomunikasi dengan SES untuk melakukan insialisasi semua sumber *inputsymbolic*. Setelah inisialisasi, MAYHEM menjalankan *binary* pada CPU di CEC. Selama eksekusi,

CEC menjalankan *dynamic* taint analysis untuk mencari *bugs*

3. Bila CEC menemukan sebuah *tainted jump condition* atau sebuah *jump target* atau sebuah *bug*, CEC akan menghentikan *concreteexecution*. Sebuah *tainted jump* berarti target tidak tergantung pada *input* dari penyerang. CEC kemudian mengirimkan instruksi kepada SES, dan SES menentukan *branch* yang mana yang mungkin ada. CEC kemudian akan menerima target *branch* selanjutnya dari SES
4. SES yang berjalan paralel dengan CEC akan menerima sebuah stream data dari *taintedinstructions* dari CEC. SES kemudian menjalankan *symbolicexecution*. SES menggunakan *path* formula dan exploitability formula.
5. Ketika MAYHEM menemukan *branch* yang memiliki *bug*, SES akan memutuskan apakah akan melanjutkan pencabangan dengan melakukan *query* pada SMT *solver*. Bila pencabangan dilanjutkan, maka cabang baru akan dikirim kepada *pathselector* untuk mendapat prioritas. Untuk memilih sebuah *path*, SES memberitahu CEC tentang adanya perubahan tersebut. Bila batasan re*source* dari sistem telah tercapai, maka *checkpoint* manager akan memilih untuk melakukan *generatecheckpoint*s daripada membuat cabang baru.
6. Selama eksekusi, SES akan berpindah antara *executor* dan *checkpoint* CEC, mengembalikan *executionstate* yang telah ada dan melanjutkan eksekusi. Untuk itu CEC membuat sebuah layer virtual untuk menangani interaksi *program* dengan sistem dan *checkpoint* antara beberapa *executionstate*s dari aplikasi.
7. Ketika MAYHEM mendeteksi adanya sebuah *taintedjumpinstruction*, maka formula eksploit akan dibangun dan *query* akan dikirimkan kepada SMT *solver* untuk menguji apakah formula tersebut cocok. Hasilnya adalah sebuah *exploit*. Bila tidak ada eksploit yang ditemukan maka SES akan terus mengeksplorasi *executionpath*.
8. Langkah diatas terus dilakukan pada setiap cabang hingga sebuah *exploit*able *bug* ditemukan, MAYHEM mencapai nilai maximum *runtime* yang ditetapkan user atau bila semua *path*s telah teruji.

4.2 Implementasi dan Pengujian

Mayhem terdiri dari sekitar 27000 baris C/C++ dan kode OCaml. Pengujian dilakukan pada 2 virtual mesin yang dijalankan pada sebuah desktop dengan prosessor 3,4 GHz Intel® Core i/-2600 CPU dan RAM 16 GB. Setiap VM memiliki 4GB RAM, menjalankan Debian Linux (Squeeze) VM dan Windows XP SP3. Pengujian dilakukan 29 *program* yang memiliki celah keamanan. *Program* yang diuji merupakan *progrkaambinary* pada Linux dan Windows, tanpa modifikasi.

Setelah beberapa menit, *onlineexecution* telah mencapai kapasitas memori maksimum dan mulai menghentikan *executionpath*. Sementara *offlineexecution* hanya membutuhkan memori yang minimal, hanya saja kecepatan pengujian tidak optimal. *Hybridexecution* pada awalnya terlihat memiliki kebutuhan memori yang mirip dengan *onlineexecution* tapi setelah beberapa menit terlihat kebutuhan memori mulai menurun. *HybridExecution* dapat melakukan penyeimbangan antara kecepatan pengujian dengan pemakaian memori.

Dari hasil pengujian MAYHEM menemukan 2 zero-day *exploit* pada 2 aplikasi Linux.. *Program* tercepat yang berhasil ditemukan eksploitnya adalah aplikasi Linux iwconfig, yaitu selama 1,9 detik. Sementara aplikasi yang membutuhkan waktu terlama untuk membuat eksploitnya adalah aplikasi Windows Dizzy, yang memerlukan waktu 4 jam

4. Simpulan

SAGE mengembangkan metode *whiteboxFuzzTesting* yang didasarkan pada pengujian *symbolicexecution* dan *dynamictestgeneration*. SAGE mengenalkan algoritma pencarian baru yang dinamakan *generational search*. Hasil pengujian menunjukkan Algoritma ini dapat menghasilkan pencarian *exploitable bugs* yang lebih baik dibandingkan dengan metode *FuzzTesting* yang menggunakan *blackboxtesting*.

Selain melakukan pencarian *exploit*able *bugs* secara otomatis, AEG juga dapat menghasilkan *exploit* yang dapat digunakan pada *bugs* yang ditemukan, AEG membutuhkan *sourcecode* dari aplikasi yang akan diuji. AEG yang dibangun dengan basis KLEE, mengenalkan teknik *preconditioned symbolicexecution*. Hasil pengujian menunjukkan AEG mampu menemukan *exploitable bugs* dan meng-*generate exploit* yang sesuai.

MAYHEM mengembangkan metode *hybridsymbolicexecution* dan *index-basedmemory* modeling sehingga dapat melakukan pencarian *exploit*able *bugs* pada *binarycode*. MAYHEM merupakan pengembangan dari AEG. Hasil pengujian menunjukkan MAYHEM mampu melakukan pencarian *exploitable bugs* dan melakukan generate exploit yang sesuai dengan waktu yang cepat dan pemakaian memori yang optimal.

Daftar Pustaka

[1] Launchpad, “Launchpad,” 19 12 2012. [Online]. Available: https://bugs.launchpad.net/ubuntu. [Diakses 19 12 2012]

[2] K. Sen et al, "CUTE: A concolic unit testing engine for C," dalam ACM Symposium on the Foundations of Software Engineering,, 2005.
[3] D. Song et.al, "BitBlaze: A New Approach to Computer Security via Binary Analysis," dalam the 4th International Conference on Information Systems Security (ICISS), 2008.
[4] C. Cadar et.al, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," dalam the USENIX Symposium on Operating System Design and Implementation, Dec. 2008.
[5] P. Godefroid et.al, "Automated whitebox fuzz testing," dalam the Network and Distributed System Security Symposium, Feb. 2008.
[6] A. V. Thakur et al., "Directed proof generation for machine code," Computer Aided Verification, vol. 22nd, p. 288–305, 2010.
[7] T. Avgerinos et.al, "AEG : Automatic exploit generation," dalam the Network and Distributed System Security Symposium, Feb. 2011.
[8] G. C. V. Chipounov et.al, "S2E: A platform for in-vivo multi-path analysis of software systems," dalam International Conference on Architectural Support for Programming Languages and Operating Systems, 2011.
[9] S. K. Cha et.al, "Unleashing Mayhem on Binary Code," dalam he 2012 IEEE Symposium on Security and Privacy, Oakland, May 2012.
[10] J.King, "Symbolic execution and program testing," Communications of the ACM, vol. 19, pp. 386-384, 1976.
[11] P. Godefroid et.al, "Automated whitebox fuzz testing," dalam the Network and Distributed System Security Symposium, Feb. 2008.
[12] T. Avgerinos et.al, "AEG : Automatic exploit generation," dalam the Network and Distributed System Security Symposium, Feb. 2011.
[13] C. Cadar et al, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," dalam the USENIX Symposium on Operating System Design and Implementation, Dec. 2008.
[14] S. K. Cha et.al, "Unleashing Mayhem on Binary Code," dalam the 2012 IEEE Symposium on Security and Privacy, Oakland, May 2012.
[15] V. Chipounov et.al, "S2E: A platform for in-vivo multi-path analysis of software systems," dalam International Conference on Architectural Support for Programming Languages and Operating Systems, 201