

Makalah EC 5010 Keamanan Sistem Informasi  
Buffer Overflow dan Beberapa Pendekatan Untuk  
Mengatasinya Dalam Level Aplikasi

Alamsyah ( 13202046 )

June 8, 2006

# Table of Contents

<b>Table of Contents</b>	<b>1</b>
<b>1 Pendahuluan</b>	<b>1</b>
1.1 Latar Belakang . . . . .	1
1.2 Tujuan . . . . .	2
1.3 Batasan Masalah . . . . .	2
1.4 Sistematika Pembahasan . . . . .	3
<b>2 Buffer Overflow</b>	<b>4</b>
2.1 Konsep Dasar <i>Buffer Overflow</i> . . . . .	4
2.2 <i>Buffer Overflow</i> pada Stack . . . . .	6
2.3 Buffer Overflow dalam bahasa C/C++ . . . . .	7
<b>3 Langkah-langkah Antisipasi</b>	<b>10</b>
3.1 Validasi Jangkauan Data . . . . .	10
3.2 Verifikasi Panjang Data . . . . .	13
3.3 Mengantisipasi Karakter-karakter Ilegal . . . . .	15
3.4 Beberapa pendekatan lain . . . . .	17
3.4.1 Canary-based Defense . . . . .	17
3.4.2 ASCII Armor Region . . . . .	18
3.4.3 Penggunaan library yang lebih aman . . . . .	19
<b>4 Penutup</b>	<b>21</b>
4.1 Kesimpulan . . . . .	21
4.2 Saran . . . . .	22
<b>Reference</b>	<b>23</b>

# Chapter 1

## Pendahuluan

### 1.1 Latar Belakang

Dalam dunia kewanaman sistem komputer dan pemrograman, istilah *buffer overflow* atau *buffer overrun* merupakan istilah yang cukup terkenal. *Buffer overflow* merupakan sebuah keadaan anomali di mana sebuah proses menunjukkan perilaku yang aneh disebabkan karena adanya data-data yang disimpan melebihi kapasitas *buffer* memorinya. Perilaku yang ditunjukkan oleh sebuah proses yang mengalami *buffer overflow* bisa jadi merupakan sebuah celah keamanan yang dapat dimanfaatkan oleh pihak-pihak yang tak bertanggung jawab.

*Buffer overflow* seringkali dieksploitasi dengan mengirimkan input-input yang sengaja dirancang untuk memicu keadaan *buffer overflow* tersebut. Input-input tersebut juga dirancang sedemikian rupa sehingga saat terjadi keadaan *buffer overflow* maka proses tersebut akan berkelakuan seperti yang diinginkan oleh sang penyerang. Bila keadaan ini telah terjadi maka bukan tidak mungkin informasi-informasi penting yang tersimpan di dalamnya bisa diambil, digunakan atau diubah sekehendak hati oleh sang penyerang.

Mengingat betapa besarnya kerugian yang dapat disebabkan oleh terjadinya *buffer overflow* maka diperlukan pengetahuan yang lebih mendalam tentangnya. Tak lupa pula diperlukan penjabaran mengenai beberapa pendekatan yang dapat dilakukan untuk mengurangi resiko terjadinya *buffer overflow* ini. Diharapkan dengan adanya pengetahuan lebih dalam mengenai hal ini maka dapat dilakukan pengamanan maksimal terhadap sistem yang kita pakai.

## 1.2 Tujuan

Tujuan dari penulisan makalah ini adalah untuk memberikan informasi lebih mendalam tentang *buffer overflow* dan dampak yang bisa diberikannya terhadap sistem atau proses yang kita gunakan. Selain itu makalah ini juga bertujuan untuk memberikan informasi mengenai beberapa pendekatan yang bisa dilakukan untuk mengatasi masalah *buffer overflow* ini. Pendekatan-pendekatan yang dilakukan di sini merupakan pendekatan dalam level aplikasi terutama dalam proses penulisan program/aplikasi.

## 1.3 Batasan Masalah

Pembahasan yang akan dilakukan dalam makalah ini memiliki batasan-batasan sebagai berikut :

1. Makalah ini akan memberikan penjelasan mengenai *buffer overflow* dilihat dari penggunaan memori sistem atau proses
2. Makalah ini akan memberikan penjelasan mengenai dampak dari *buffer overflow* hanya dalam level aplikasi

3. Makalah ini akan menjabarkan beberapa pendekatan yang bisa dilakukan untuk mengatasi masalah dalam level aplikasi
4. Pada pembahasannya dalam level aplikasi maka bahasa pemrograman yang dijadikan contoh adalah bahasa C dan turunannya karena bahasa ini memiliki vulnerabilitas yang tinggi terhadap serangan *buffer overflow*.

## 1.4 Sistematika Pembahasan

Adapun urutan pembahasan yang akan dijabarkan dalam makalah ini adalah sebagai berikut :

- Pengenalan terhadap ancaman *buffer overflow*.
- Dampak atau akibat yang bisa disebabkan oleh masalah ini
- Pendekatan yang bisa dilakukan untuk mencegah masalah ini

# Chapter 2

## Buffer Overflow

### 2.1 Konsep Dasar *Buffer Overflow*

Seperti telah disebutkan pada bagian awal makalah ini, *buffer overflow* sering terjadi saat data ditulis ke dalam sebuah *buffer* sebagai tempat penampungan sementara sebelum dilakukan proses lebih lanjut terhadap data-data tersebut. Karena tidak adanya mekanisme pengecekan batas *buffer* maka jika data yang dimasukkan melebihi kapasitas *buffer* maka data-data tersebut tetap akan dimasukkan ke *buffer* dan bagian data yang melebihi ukuran *buffer* akan ditempatkan di blok memori setelah *buffer* tersebut. Seringkali jika alamat memori yang bersebelahan dengan *buffer* tersebut juga dipakai oleh program lain, maka adanya data-data *overflow* yang masuk akan merusak program tersebut. Hal ini sangat umum terjadi pada proses mengkopi *string* atau karakter dari suatu *buffer* ke *buffer* yang lain. Agar konsep mengenai *buffer overflow* dapat lebih mengerti maka kita gunakan contoh seperti **gambar 2.1**.

Pada **gambar 2.1** kita memiliki dua buah *buffer* yang kita sebut sebagai *buffer A* dan *buffer B*. *Buffer A* dialokasikan oleh program untuk menempati delapan blok memori ( asumsikan satu blok sama dengan 1 *byte* ) dan *buffer B* menempati 2 blok memori. Pada kondisi awal *buffer A* memiliki isi 0 pada delapan blok memorinya

A	A	A	A	A	A	A	A	B	B
0	0	0	0	0	0	0	0	0	3

Figure 2.1: Buffer

A	A	A	A	A	A	A	A	B	B
0	8	1	7	2	3	2	0	1	4

Figure 2.2: Buffer overflow

yang menandakan bahwa data pada *buffer* tersebut bernilai 0. Sementara di saat yang bersamaan *buffer* B yang merupakan variabel di bagian program yang lain memiliki isi 0 dan 3 yang artinya variabel pada *buffer* B bernilai 3. Suatu saat program meminta input dari pengguna dan input tersebut akan dimasukkan ke dalam *buffer* A. Jika si pengguna ternyata memasukkan input misalnya 0817232014 maka pada bagian *buffer* A dan B akan terlihat seperti **gambar 2.2**.

Pada kasus di atas input yang dimasukkan oleh pengguna terdiri atas 10 karakter sedangkan *buffer* yang disediakan hanya untuk delapan karakter. Disebabkan tidak

adanya mekanisme *bound checking* maka akhirnya input yang berlebih itu dimasukkan juga ke blok memori yang bersebelahan yaitu *buffer* B. Hal ini menyebabkan *buffer* B yang semula bernilai 3 berubah nilainya menjadi 14. Bila nilai yang berada pada *buffer* B ini mengendalikan suatu proses dalam program maka akibat perubahan ini bisa saja program tersebut melakukan proses yang tidak semestinya dilakukan dan hal inilah yang umum disebut sebagai *buffer overflow attack*.

## 2.2 *Buffer Overflow* pada Stack

Selain dapat mengubah nilai suatu variabel, *buffer overflow* juga dapat digunakan untuk mengeksekusi kode-kode yang disusupkan oleh si penyerang saat program yang bersangkutan berjalan. Pada kasus seperti ini, serangan *buffer overflow* memanfaatkan perilaku program yaitu setiap memanggil suatu fungsi selalu menempatkan suatu *return address* pada puncak *stack* dalam memori. Penyerang mengeksploitasi situasi ini dengan menyusupkan suatu kode yang akan menyebabkan *overflow*. Kode yang disusupkan penyerang ini akan menyebabkan *overflow* dan mengubah *return address* yang telah disimpan dalam puncak *stack* sebelum fungsi dipanggil. Akibatnya saat fungsi selesai dijalankan maka program akan kembali ke *return address* yang telah diubah si penyerang yang kemungkinan besar adalah *malicious code* yang telah dibuat oleh si penyerang. Keadaan seperti ini dinamakan *stack overflow* dan *heap overflow*. Ilustrasi untuk keadaan ini dapat dilihat pada **gambar 2.3**.

Pada **gambar 2.3**, C merupakan alokasi memori untuk program utama. Program C akan memanggil fungsi A sehingga program C secara otomatis akan menyimpan *return address* pada puncak *stack* yaitu di B. Fungsi A yang dipanggil merupakan program yang menyebabkan *overflow* dan saat dipanggil A akan merusak batas *buffer*

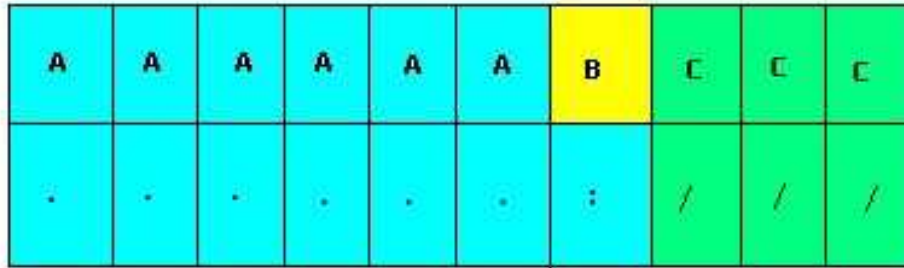


Figure 2.3: Stack overflow

dan mengubah nilai B yang merupakan *return address*. Akibatnya saat fungsi A selesai dilakukan maka program akan kembali ke *return address* yang salah.

## 2.3 Buffer Overflow dalam bahasa C/C++

Diantara bahasa pemrograman yang umum dipakai oleh *programmer* pada masa kini mungkin bahasa C dan turunannya merupakan bahasa yang cukup banyak digunakan. Bahasa C menurut sebagian orang merupakan bahasa tingkat menengah yang diartikan sebagai bahasa pemrograman tingkat tinggi dengan kemampuan bahasa pemrograman tingkat rendah. Hal ini dimungkinkan karena sifat bahasa C dan turunannya yang mempunyai sintaks seperti bahasa manusia ( dalam hal ini bahasa Inggris tentunya ) namun memiliki kemampuan dalam mengakses perangkat keras secara langsung ( dalam hal ini memori ) dengan menggunakan konsep pointer ( \* )

Bahasa pemrograman pada umumnya jarang mempunyai masalah dengan *buffer*

*overflow* disebabkan karena pada saat mereka ingin melakukan penulisan ke memori mereka selalu memastikan bahwa semuanya selalu dalam batas memori yang dialokasikan. Namun sayangnya tidak demikian dengan bahasa pemrograman C dan turunannya. Kemampuan C dalam mengakses perangkat keras secara langsung menyebabkan kemungkinan untuk terjadinya *buffer overflow* sangat besar terutama saat berurusan dengan masalah *string*. Dalam C, *string* dianggap sebagai *array of character* sehingga bahasa C tidak pernah mengatur batasan tentang panjang data *string* ini. Bahasa C hanya mengenal batasan suatu *string* jika suatu saat ditemukan karakter NULL dalam string ( *NULL terminates* ). Jika kondisi ini tidak ditemukan maka kemungkinan besar terjadinya *overflow* akan sangat besar.

Salah satu contoh dari fungsi standar dalam ANSI C library adalah sebagai berikut:

```
char *gets(char *str);
```

Fungsi `gets()` di atas membaca data dari standar input dan memasukkannya ke dalam bagian memori yang ditunjuk oleh `str` sampai ditemukan adanya baris baru atau sampai ditemukannya tanda akhir file ( *EOF* ). Dalam kasus ini *buffer overflow* akan sangat mudah dilakukan cukup dengan menghilangkan/menghindari tanda baris baru atau akhir file sehingga fungsi akan membaca terus data dari standar input sampai batas yang tak tentu.

Fungsi-fungsi lain yang berkenaan dengan operasi *string* pada bahasa C ( misalnya: `strcpy()` atau `strncpy()` ) pada umumnya tidak melibatkan proses *bound checking* untuk memeriksa apakah alokasi memori yang disediakan akan mencukupi untuk data yang akan dimasukkan. Ditambah lagi pada umumnya operasi yang bersangkutan

dengan tipe data string hanya mengandalkan karakter NULL sebagai batasan ukuran dari sebuah string sehingga jika karakter tersebut kebetulan hilang karena suatu proses yang tidak sempurna maka string akan menjadi sesuatu yang tak hingga dan akan menyebabkan *overflow* ke daerah memori yang berdekatan.

Mengingat betapa besarnya celah untuk melakukan *buffer overflow* pada aplikasi-aplikasi yang ditulis dengan bahasa C dan turunannya maka pada pembahasan di bagian berikutnya akan dibicarakan beberapa pendekatan yang bisa dilakukan untuk mengatasi masalah ini. Penjabaran ini dimaksudkan agar para *programmer* pada umumnya dan pemakai bahasa C dan turunannya pada khususnya dapat mengetahui konsep-konsep umum mengamankan aplikasi mereka dari ancaman *buffer overflow*.

# Chapter 3

## Langkah-langkah Antisipasi

Pada bagian-bagian sebelumnya dari makalah ini telah diberikan penjelasan mengenai konsep dasar dari serangan *buffer overflow* dan apa saja dampak dari serangan tersebut terhadap aplikasi atau sistem yang kita miliki. Dalam bagian sebelumnya juga telah disebutkan bahwa salah satu bahasa pemrograman yang sangat rentan terhadap serangan *buffer overflow* adalah bahasa C dan turunannya terutama untuk hal-hal yang berkenaan dengan operasi data *strings*. Pada bagian ini akan disajikan beberapa pendekatan yang dapat dilakukan untuk mengatasi masalah *buffer overflow* pada aplikasi yang kita buat. Pendekatan-pendekatan ini tidak menjanjikan bahwa aplikasi anda bebas 100 persen dari bahaya *buffer overflow* namun setidaknya dengan melakukannya beberapa bagian dari aplikasi yang dibuat sudah dapat diamankan.

### 3.1 Validasi Jangkauan Data

Saat kita menggunakan bahasa pemrograman tertentu maka pastinya kita akan dihadapkan dengan berbagai tipe data yang masing-masing memiliki jangkauan tertentu. Jangkauan nilai yang ditawarkan oleh tipe-tipe data tersebut merupakan sesuatu yang dapat didukung oleh *hardware* dan "dianggap cukup" untuk memenuhi

kebutuhan kita. Sebagai contoh jika kita menggunakan tipe data integer 32 bit maka kita dapat memasukkan nilai mulai dari -2.147.483.648 sampai 2.147.483.647 yang dianggap komputer cukup untuk memenuhi kebutuhan kita. Pada kenyataannya terdapat beberapa program yang menemukan bahwa jangkauan nilai ini terkadang tidak cukup untuk memenuhi kebutuhannya.

Di saat terdapat kemungkinan di mana aplikasi yang anda rancang akan menerima input dari pengguna berupa nilai yang tidak seharusnya ( melebihi jangkauan tipe data ) maka ada baiknya jika kita menambahkan sepotong kode program untuk memeriksa/memvalidasi input yang diberikan pengguna sebelum diproses lebih lanjut. Jika proses validasi gagal maka program akan mengeluarkan peringatan/ pesan *error* atau mungkin program akan dihentikan secara otomatis. Salah satu contoh kode untuk melakukan validasi yang ditulis dalam bahasa C++ dapat dilihat pada **gambar 3.1**

Kode pada **gambar 3.1** pada awalnya melakukan konversi input ke dalam bentuk `Int32` dengan menggunakan method `Parse()`.Selanjutnya program juga akan melakukan pemeriksaan nilai apakah terlalu besar atau terlalu kecil menggunakan `System::OverflowException` dan pemeriksaan terhadap nilai-nilai dengan format yang tidak benar dengan menggunakan `System::FormatException`. Setelah semuanya dilakukan baru kemudian program melakukan pemeriksaan nilai input terhadap jangkauan yang ditentukan yaitu 1 sampai 40.000. Jika nilai input berada di luar jangkauan tersebut maka akan dikeluarkan *error message* yang memperingatkan pengguna bahwa input yang dimasukkannya salah.

```
System::Void btnDataRange_Click(System::Object * sender,
                               System::EventArgs * e)
{
    Int32 TestData; // Holds the input value.

    try
    {
        // Always attempt to parse the data first.
        TestData = Int32::Parse(txtInput1->Text);
    }
    catch (System::OverflowException *OE)
    {
        // React to the overflow error.
        MessageBox::Show("Type a value between 1 and 40,000.",
                        "Input Error",
                        MessageBoxButtons::OK,
                        MessageBoxIcon::Error);
        return;
    }
    catch (System::FormatException *FE)
    {
        // React to the overflow error.
        MessageBox::Show("Type the number without extra charaters.",
                        "Input Error",
                        MessageBoxButtons::OK,
                        MessageBoxIcon::Error);
        return;
    }

    // Test the specific data range.
    if (TestData < 1 || TestData > 40000)

        // React to the data range error.
        MessageBox::Show("Type a value between 1 and 40,000.",
                        "Input Error",
                        MessageBoxButtons::OK,
                        MessageBoxIcon::Error);
}
```

Figure 3.1: Validasi Jangkauan Data

## 3.2 Verifikasi Panjang Data

Seperti telah diterangkan pada bagian sebelumnya, bahasa pemrograman C dan turunannya seringkali bermasalah jika harus bekerja dengan tipe data String. String pada bahasa C dianggap sebagai *array of Character* sehingga tidak ada yang membatasi panjangnya kecuali diterminasi dengan karakter NULL atau dibatasi dengan kemampuan *hardware* sistem itu sendiri.

Dalam imlementasinya pada sebuah program, jarang sekali dibutuhkan panjang string yang tak terhingga. Pengembang program pada umumnya hanya membutuhkan alokasi string dalam jangkauan tertentu agar programnya dapat bekerja dengan baik dan sesuai harapan. Oleh karena itu untuk menghindari serangan *buffer overflow* ada baiknya dalam sebuah program ditentukan secara pasti berapa batas maksimum dan juga batas minimum dari jumlah string yang akan dipakai. Selanjutnya terhadap input yang dimasukkan dilakukan pemeriksaan apakah input memenuhi syarat jumlah karakter yang digunakan atau tidak. Jika jumlah karakter memenuhi syarat maka program dapat dilanjutkan, sebaliknya jika panjang karakter tak memenuhi syarat maka pengguna akan diberi *error message* atau program dihentikan. Contoh implementasi metode ini dalam bahasa C++ dapat dilihat pada **gambar 3.2**

Pada kode program di **gambar 3.2** program melakukan validasi dengan method `ProcessData()`. Method tersebut menerima input berupa string input, batas maksimum panjang string dan batas minimum panjang string. Sebelum melakukan pemeriksaan terhadap input terlebih dahulu dilakukan pemeriksaan terhadap batas

```

System::Boolean ProcessData(String *Input,
                             Int32 UpperLimit,
                             Int32 LowerLimit)
{
    StringBuilder *ErrorMsg; // Error message.

    // Check for an input error.
    if (UpperLimit < LowerLimit)
    {
        // Create the error message.
        ErrorMsg = new StringBuilder();
        ErrorMsg->Append($"The UpperLimit input must be greater than ");
        ErrorMsg->Append($"the LowerLimit number.");

        // Define a new error.
        System::ArgumentException *AE;
        AE = new ArgumentException(ErrorMsg->ToString(),
                                   S"UpperLimit");

        // Throw the error.
        throw(AE);
    }

    // Check for a data length error condition.
    if (Input->Length < LowerLimit || Input->Length > UpperLimit)
    {
        // Create the error message.
        ErrorMsg = new StringBuilder();
        ErrorMsg->Append($"String is the wrong length. Use a string ");
        ErrorMsg->Append($"between 4 and 8 characters long.");

        // Define a new error.
        System::Security::SecurityException *SE;
        SE = new SecurityException(ErrorMsg->ToString());

        // Throw the error.
        throw(SE);
    }

    // If the data is correct, return true.
    return true;
}

System::Void btnDataLength_Click(System::Object * sender,
                                 System::EventArgs * e)
{
    try
    {
        // Process the input text.
        if (ProcessData(txtInput2->Text, 8, 4))

            // Display a result message for correct input.
            MessageBox::Show(txtInput2->Text,
                              "Input String",
                              MessageBoxButtons::OK,
                              MessageBoxIcon::Information);
    }
    catch (System::Security::SecurityException *SE)
    {
        // Display an error message for incorrect input.
        MessageBox::Show(SE->Message,
                          "Input Error",
                          MessageBoxButtons::OK,
                          MessageBoxIcon::Error);
    }
    catch (System::ArgumentException *AE)
    {
        // Display an error message for incorrect input.
        MessageBox::Show(AE->Message,
                          "Argument Error",
                          MessageBoxButtons::OK,
                          MessageBoxIcon::Error);
    }
}

```

Figure 3.2: Verifikasi Panjang Data

maksimum dan batas minimum. Jika batas maksimum yang dimasukkan lebih rendah dari batas minimum maka akan dikeluarkan sebuah pesan *error* untuk memperingatkan pengguna.

Setelah dilakukan validasi terhadap batas maksimum dan batas minimum berikutnya dilakukan validasi terhadap input itu sendiri. Jika input memiliki jumlah karakter terlalu banyak atau terlalu sedikit maka program akan memanggil sebuah *exception* `System::Security::SecurityException`. Metode ini akan memberikan peringatan kepada pengguna bahwa terdapat kesalahan pada input yang dimasukkan. Pada kasus ini program hanya memberikan peringatan kepada pengguna tetapi masih memberikan kesempatan pada pengguna jika masih ingin menjalankan program dengan kondisi *buffer overflow*. Dalam implementasi yang lebih baik, *exception* yang diberikan mungkin saja bisa berupa pemberhentian program secara otomatis atau hal-hal lain yang bisa dilakukan untuk mencegah terjadinya *buffer overflow*.

### 3.3 Mengantisipasi Karakter-karakter Ilegal

Cara yang paling sederhana dalam menguji keamanan sebuah sistem atau aplikasi yaitu dengan memasukkan input berupa karakter-karakter yang merupakan bagian dari program tersebut (misalnya `:ls` pada UNIX/Linux, `$` pada php, `cout<<jj` pada C++ dan sebagainya). Jika aplikasi yang dimasukkan input tersebut ternyata mengeksekusi potongan-potongan perintah tersebut maka artinya satu celah keamanan telah terbuka. Untuk mengeksploitasi aplikasi tersebut lebih lanjut sang penyerang hanya perlu merumuskan perintah-perintah apa yang seharusnya dimasukkan untuk mencapai tujuannya. Serangan jenis ini umumnya dialami oleh aplikasi-aplikasi berbasis web namun tidak menutup kemungkinan untuk terjadi dalam aplikasi desktop.

```

System::Boolean CheckChars(System::String *Input)
{
    StringBuilder *ErrorMsg; // Error message.
    Regex      *R;          // Regular expression.

    // Create a regular expression for match purposes.
    R = new Regex("[A-Za-z]");

    // Check for a data length error condition.
    if (R->Matches(Input)->Count < Input->Length)
    {
        // Create the error message.
        ErrorMsg = new StringBuilder();
        ErrorMsg->Append($"String contains incorrect characters. ");
        ErrorMsg->Append($"Use only A through Z and a through z.");

        // Define the exception.
        System::Security::SecurityException *SE;
        SE = new SecurityException(ErrorMsg->ToString());

        // Throw the exception.
        throw(SE);
    }

    // If the data is correct, return true.
    return true;
}

```

Figure 3.3: Menggunakan Regular Expression

Untuk mengatasi masalah-masalah akibat adanya karakter ilegal yang dimasukkan maka diperlukan mekanisme untuk memeriksa input apakah mengandung karakter-karakter yang tidak diperbolehkan atau tidak. Masing-masing bahasa pemrograman umumnya memiliki mekanisme tersendiri dalam memeriksa masalah ini. Salah satu contohnya dapat dilihat pada **gambar 3.3** yang merupakan mekanisme pemeriksaan karakter input menggunakan bahasa C++.

Pada contoh tersebut digunakan objek RegEx untuk menentukan karakter-karakter apa saja yang diterima oleh program sebagai inputnya. Dalam kasus ini input yang diterima hanyalah karakter A sampai Z dan a sampai z ( bahkan karakter *space* pun tidak diperbolehkan). Selain dapat digunakan untuk memvalidasi karakter, objek RegEx juga dapat digunakan untuk memvalidasi beberapa pola seperti nomor telepon untuk pemakaian yang lebih luas dan fleksibel. Jika karakter yang dimasukkan oleh pengguna tidak termasuk dalam karakter yang divalidasi maka seperti biasa program akan mengirimkan pesan error yang memperingatkan pengguna bahwa input yang dimasukkannya salah ( tidak valid ).

## 3.4 Beberapa pendekatan lain

Beberapa pendekatan untuk mencegah terjadinya *buffer overflow* pada level aplikasi telah dibicarakan di atas. Bagian berikut ini akan memberikan beberapa pendekatan tambahan yang sifatnya lebih rumit. Pendekatan yang disajikan di bawah ini diambil dari sudut pandang kompiler dan sistem secara umum namun sifatnya tetap mendukung keamanan aplikasi yang akan kita jalankan nantinya.

### 3.4.1 Canary-based Defense

Metode *Canary-based defense* dicetuskan oleh seorang peneliti bernama Crispin Cowan melalui pendekatan yang disebut **StackGuard**. Konsep dari pendekatan ini yaitu memodifikasi kompiler C sehingga saat dilakukan kompilasi maka ada sebuah nilai yang ditambahkan di depan *return address* yang disebut "canary" *value*. Saat sebuah fungsi telah selesai dijalankan dan akan kembali ke *return address* maka terlebih dahulu nilai "canary" tersebut akan diperiksa. Jika terjadi serangan *buffer overflow*

yang mengubah nilai *return address* maka nilai "canary" juga akan berubah karenanya. Nilai "canary" yang berubah akan segera diketahui program dan program akan dihentikan secara otomatis.

Perlindungan dengan metoda canary sangat bermanfaat untuk mencegah terjadinya serangan *buffer overflow* pada stack (*stack overflow*). Sayangnya metoda ini tidak dapat melindungi aplikasi dari serangan-serangan *buffer overflow* lain seperti serangan *buffer overflow* yang mengubah nilai variabel akibat letaknya berdampingan dengan *buffer* variabel lain yang *overflow*. Meskipun demikian, konsep ini terus dikembangkan sehingga diharapkan kemampuannya dapat diperluas untuk mengatasi jenis serangan *buffer overflow* yang lain.

Metode ini meskipun masih memiliki beberapa kelemahan telah diadopsi oleh beberapa kompiler C. Pengembangan dari konsep **StackGuard** yaitu *ProPolice* / *Stack Smashing Protector (SSP)* telah diimplementasikan pada GNU C Compiler (*gcc*) yang terdapat pada distribusi OpenBSD semenjak bulan Mei 2003. Selain itu perusahaan Microsoft juga telah menambahkan sebuah *flag* kompiler (*/GS*) sebagai implementasi konsep canary pada kompiler C yang dirilisnya.

### 3.4.2 ASCII Armor Region

Pendekatan ini didasari suatu ide yaitu membuat eksekusi kode saat berada dalam stack menjadi tidak mungkin. Ide ini mengalami banyak masalah dalam implementasinya, terlebih lagi ide ini sangat sulit diimplementasikan dalam mesin-mesin berprosesor x86 (prosesor yang umum digunakan saat ini).

Setelah dilakukan penelitian lebih lanjut maka muncul pendekatan yang lebih masuk akal untuk diimplementasikan yaitu memindahkan semua kode yang *executable*

ke dalam area memori yang disebut ”**ASCII armor region**”. Konsep ini didasari pemikiran bahwa kebanyakan penyerang yang memanfaatkan *buffer overflow* mengalami kesulitan jika harus membuat sebuah program kembali ke sebuah *return address* dengan angka 0 di dalamnya. Oleh karena itu dengan memindahkan kode-kode program yang *executable* ke dalam sebuah area memori di mana terdapat alamat-alamat yang memiliki angka 0 akan lebih mempersulit si penyerang jika ingin merusak program dengan serangan *buffer overflow*.

Namun pendekatan *ASCII armor* ini tetap saja memiliki beberapa kelemahan. Daerah *ASCII armor* yang terentang antara alamat memori 0 sampai 0x01010100 terkadang tidak mencukupi untuk program-program berukuran besar sehingga perlindungannya tidak akan terlalu efektif. Meskipun demikian konsep ini tetap dipakai oleh beberapa distro Linux di antaranya Fedora Core (versi gratis dari Red Hat) dan OpenWall GNU/Linux(OWL) sebagai pengaman dari serangan *buffer overflow*.

### 3.4.3 Penggunaan library yang lebih aman

*Library* standar bahasa C yang memungkinkan terjadinya operasi pada strings merupakan celah untuk terjadinya *buffer overflow*. Untuk lebih meningkatkan keamanan aplikasi yang kita buat maka akan lebih baik jika kita menggunakan *library* yang lebih baik dan mendukung perlindungan terhadap *buffer overflow*.

Salah satu *library* yang dapat digunakan adalah **SafeStr** *library*. **SafeStr** menyediakan implementasi dari ukuran string yang dapat berubah-ubah dalam bahasa C. *Library* ini juga menyediakan mekanisme perhitungan alokasi dan ukuran sebenarnya dari setiap string. Jika terjadi hal-hal yang menyebabkan ukuran string bertambah melebihi alokasi memori untuk string tersebut maka *library* ini secara otomatis akan

menaikkan pula alokasi memori untuk string sekurang-kurangnya sampai ukurannya menyamai ukuran string. Karena pengaturannya yang dinamis inilah maka penggunaan *library* **SafeStr** dianggap dapat mengurangi resiko terjadinya *buffer overflow* pada aplikasi.

# Chapter 4

## Penutup

### 4.1 Kesimpulan

Berdasarkan pembahasan terhadap *buffer overflow* yang telah diberikan dalam makalah ini maka dapat diambil beberapa kesimpulan sebagai berikut:

1. Masalah *buffer overflow* merupakan masalah yang masih sering terjadi saat ini dan sampai saat ini masih dilakukan penelitian tentang cara yang efektif untuk menangani masalah ini
2. *Buffer overflow* dapat menyebabkan aplikasi atau program yang kita buat berkelakuan tidak semestinya. Hal ini sering dimanfaatkan oleh beberapa orang untuk menyerang sistem sehingga sistem dapat berkelakuan sesuai keinginan penyerang.
3. Bahasa pemrograman C dan turunannya merupakan bahasa pemrograman yang paling rentan terhadap ancaman *buffer overflow* terutama bila berkaitan dengan masalah operasi string
4. Terdapat beberapa pendekatan yang bisa digunakan untuk mengatasi masalah

*buffer overflow* baik itu dari aplikasi itu sendiri, kompiler, library yang digunakan ataupun sistem operasi.

5. Meskipun demikian pendekatan-pendekatan yang digunakan tidak menjamin sistem aman 100 persen. Masih ada celah-celah lain yang mungkin belum tereksploitasi atau belum ditemukan solusi untuk mencegahnya.

## 4.2 Saran

- Dalam membangun sebuah aplikasi seharusnya faktor keamanan menjadi salah satu faktor yang diperhitungkan selain faktor fungsional
- Pemilihan bahasa pemrograman juga seharusnya menjadi pertimbangan terlebih lagi jika ingin memakai bahasa pemrograman yang rentan terhadap masalah *buffer overflow*
- Jika memang dibutuhkan menggunakan bahasa pemrograman tersebut maka gunakanlah metod-metode pencegahan yang telah dibicarakan. Mungkin akan menurunkan performa namun keamanan aplikasi akan lebih terjamin

# Reference

1. Huang, Yinrong,"Protection Against Exploitation of Stack and Heap Overflows",Exurity.Inc,Canada:2003.
2. Meisser, Matt and John Viega,"Secure Programming Cookbook for C and C++", O.Reilly,California:2003.
3. Mueller,Jhon,"Preventing Buffer Oveflow in Visual C++ Applications", 2004.
4. Wheeler,David,"Secure programmer : Countering buffer overflow", 2004. Artikel dapat didownload di :[ww128.ibm.com/developerworks/linux/library/l-sp4.html](http://www128.ibm.com/developerworks/linux/library/l-sp4.html)
5. <http://www.searchappsecurity.techtarget.com/originalContent>.
6. <http://www.windowsecurity.com/articles>