

BUDI RAHARDJO

KEAMANAN PERANGKAT LUNAK

Contents

<i>Pendahuluan</i>	7
<i>Dasar-Dasar Keamanan</i>	13
<i>Secure Software Development Life Cycle</i>	17
<i>Security Requirement</i>	21
<i>Secure Design</i>	25
<i>Secure Coding</i>	27
<i>Pengujian</i>	29
<i>Penutup</i>	35
<i>Bibliography</i>	37
<i>Mengenai Penulis</i>	39

Kata Pengantar

Buku ini merupakan buku pegangan untuk kuliah “Keamanan Perangkat Lunak” (Software Security) yang saya ajarkan di Institut Teknologi Bandung (ITB). Ketika saya mengajarkan kuliah ini tahun lalu, sayangnya belum ada buku pegangan dalam Bahasa Indonesia. Bahkan buku teks mengenai hal ini dalam Bahasa Inggris pun masih dapat dikatakan jarang. “Terpaksa” buku ini harus dibuat.

Selain buku, tools untuk mengajarkan ilmu ini juga masih belum banyak. Ini merupakan masalah lain. Untuk sekarang, kita bereskan kekurangan bukunya dahulu.

Fokus dari pembahasan buku ini adalah pada keamanan dari perangkat lunak. Dasar-dasar dari keamanan tidak akan dibahas secara panjang lebar pada buku ini. Pembaca diharapkan dapat membaca dari sumber lainnya.

Buku ini masih dalam tahap pengembangan. Pembaharuan akan masih sering berlangsung. Untuk itu saya menyarankan agar buku ini tersedia dalam format elektronik sehingga mengurangi kebutuhan kertas untuk mencetaknya. Selain itu format elektronik juga mempermudah distribusi buku ini. Jika Anda ingin menyediakan buku ini secara online di tempat Anda (misalnya Anda mengajarkan kuliah yang sama), hubungi saya agar versi di tempat Anda sama barunya dengan versi yang ada di saya. Ini adalah versi 0.1.

Semoga buku ini bermanfaat.

Bandung, Agustus 2013 - Februari 2015

Penulisan daftar pustaka: Budi Rahardjo, “*Keamanan Perangkat Lunak*”, PT Insan Infonesia, 2014.

Pendahuluan

PERANGKAT LUNAK (software) sudah menjadi bagian dari kehidupan kita sehari-hari. Bahkan dapat dikatakan sebagian dari sudah bergantung kepada perangkat lunak.

Berapa banyak di antara kita yang masih mengambil uang melalui kantor cabang? Sementara itu, berapa kali kita sudah mengunjungi mesin ATM dalam satu bulan terakhir? Dapatkah kita hidup tanpa kenyamanan mesin ATM? Semestinya jawabannya adalah iya, tetapi siapa di antara kita yang mau menjadi nasabah sebuah bank yang tidak memiliki layanan ATM? Maukah Anda? Di belakang layanan ATM ini ada perangkat lunak yang menjalankannya.

Pemesanan tiket pesawat terbang sekarang banyak yang dilakukan dengan menggunakan web. Penerbangan Air Asia, misalnya, memiliki situs AirAsia.com yang dapat digunakan untuk memesan tiket. Tiketpun tidak harus berbentuk fisik tiket yang dicetak dengan kertas khusus. Tiket dapat kita cetak sendiri dengan menggunakan printer kita. Bahkan kita dapat hanya menunjukkan berkas tiket tersebut dengan menggunakan perangkat *tablet*, misalnya.

Pemesanan tiket kereta api pun sekarang dapat dilakukan melalui gerai Alphasmart dengan menggunakan software yang sudah tersedia di sana. Kalau kita ke stasiun kereta api pun, penjualan tiket juga sudah menggunakan software. Sesampainya di stasiun kereta api, kertas tiket dapat dicetak sendiri oleh pengguna dengan menggunakan terminal yang tersedia di sana. Ini software.

Situs tiket.com menyediakan layanan pembelian tiket pesawat, kereta api, dan sebagainya secara online. Saya sering menggunakan layanan ini. Ini jelas-jelas menggunakan software.

Saat ini telah beredar ratusan juta handphone di Indonesia. Kebanyakan handphone yang beredar berupa *smartphone* yang berbasis sistem operasi Android. Selain Android ada banyak juga handphone iPhone yang menggunakan sistem operasi iOS. Aplikasi untuk handphone tersebut dapat ditambahkan sendiri oleh pengguna melalui toko (*store*) yang tersedia untuk masing-masing sistem operasi tersebut, misalnya Playstore untuk Android dan Apple Store

^o Mesin ATM yang kita gunakan ada yang menggunakan sistem operasi khusus, tetapi ada juga yang menggunakan sistem operasi umum - yang sudah diperkuat (*hardened*). Sementara itu aplikasi di belakangnya, *core banking*, menggunakan berbagai jenis sistem operasi dan aplikasi. Kesemuanya sangat bergantung kepada perangkat lunak.

untuk iPhone. Ini semua merupakan software.

Perangkat yang terkait dengan kesehatan juga menggunakan software. Bahkan *devices* yang berbentuk perangkat keras (*hardware*) (*embedded system*) sebetulnya di dalamnya memiliki software juga yang disebut *firmware*.

Kegagalan Perangkat Lunak

Bagaimana jika perangkat lunak ini gagal beroperasi? Apa efeknya? Kegagalan perangkat lunak dapat berakibat ketidaknyamanan, kerugian finansial, hilangnya nama baik, timbulnya masalah kesehatan, dan bahkan sampai kepada hilangnya nyawa. Cerita mengenai kegagalan software dapat dibaca pada bukunya Ivars Peterson ¹.

Sebagai contoh, jika sebuah mesin ATM tidak berfungsi, maka kita akan kesal dan mencari mesin ATM lainnya. Kegagalan ini hanya menimbulkan ketidaknyamanan semata ². Lain ceritanya jika aplikasi *core banking* yang mencatat saldo rekening bank kita gagal berfungsi dan menihilkan saldo kita, maka ini bukan hanya membuat kesal tetapi menjadi masalah finansial yang sangat besar bagi semua pihak. Berapa kerugian yang terjadi akibat gagalnya perangkat lunak? Lebih jauh lagi silahkan baca buku Ivars Peterson ³.

Jika keagalaman perangkat lunak terjadi pada alat pacu jantung, misalnya, dapat dibayangkan masalah yang ditimbulkannya. Ini bukan lagi masalah finansial, tetapi masalah nyawa. Berapa harga yang ingin kita pasang untuk sebuah nyawa?

Kegagalan perangkat lunak umumnya dilihat dari kacamata fungsional, yaitu aplikasi tidak berfungsi seperti yang diharapkan. Namun kegagalan perangkat lunak dapat juga menimbulkan masalah keamanan (*security*), seperti misalnya orang yang tidak berhak mengakses rekening kita ternyata dapat membaca dan mengubah data.

Sebagai contoh, ada kasus *bug* dalam sistem operasi iOS 7 yang digunakan oleh Apple dalam produk iPhone-nya. Pada gambar 1 ⁴ ditunjukkan kode dari software yang bertujuan untuk menguji apakah sertifikat SSL yang digunakan valid. Jika kita perhatikan lebih lanjut ada kesalahan, yaitu duplikat "goto fail;". Baris "goto fail;" yang kedua akan selalu dieksekusi karena tidak ada bagian "if"-nya sehingga kode-kode setelah baris itu tidak akan dicek. Program akan selalu pergi ke "goto fail". Akibat dari kesalahan ini diduga banyak pihak dapat masuk ke perangkat iPhone dengan sertifikat palsu. Kesalahan ini sudah diperbaiki di iOS 7.0.6.

Contoh bug di software yang baru saja diketahui ketika buku ini ditulis (September 2014) adalah bug di shell *bash*, yang dikenal juga dengan istilah *shellshock*. Pada bug ini, kita dapat menyisipkan perintah-perintah di belakang *assignment* pada *environmental variable*

¹ Ivars Peterson. *Fatal Defect: Chasing Killer Computer Bugs*. Random House, 1995

² Richard Miller. Social network analysis. *IEEE Transaction of Networks*, 2013

³ Ivars Peterson. *Fatal Defect: Chasing Killer Computer Bugs*. Random House, 1995

⁴ Sumber:
<http://www.wired.com/threatlevel/2014/02/gotofail/>


```

static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                uint8_t *signature, UInt16 signatureLen)
{
    OSStatus      err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}

```

Figure 1: Kesalahan kode yang mengakibatkan alur logika salah

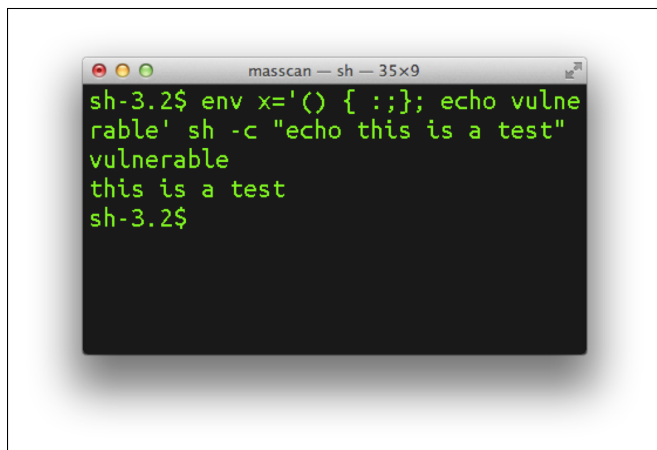
yang kemudian akan dieksekusi oleh bash. Seharusnya perintah-perintah di belakang assignment itu tidak diterima. Dalam contoh pada dalam gambar 2, perintah tersebut hanya *echo vulnerable* saja. Perintah yang disisipkan (oleh penyerang) boleh jadi perintah yang lebih berbahaya, misalnya "rm -rf /" (yang menghapus seluruh direktori *slash*). Jika eksekusi bash ini menggunakan level *root*, misal merupakan bagian dari shellsript di sebuah server web yang dijalankan dengan akun *root*, maka hasilnya dapat fatal.

Masalah security seperti ini dikaitkan dengan akses atau secara umum dapat dimasukkan ke dalam kategori aspek *confidentiality* (kerahasiaan data) dan *integrity* (integritas data). Selain dari itu kegagalan perangkat lunak dapat menyebabkan aplikasi menjadi gagal berfungsi (mati, hang, crash, atau terlalu lambat). Yang ini dikaitkan dengan aspek *availability* (ketersediaan). Aspek-aspek keamanan ini yang akan menjadi fokus pembahasan dengan bahasan lebih ke aspek perangkat lunaknya. Selain sisi perangkat lunak ada juga sisi jaringan, tetapi yang itu menjadi bahasan terpisah.

Jumlah kelemahan perangkat lunak semakin meningkat, sebagaimana ditunjukkan pada Gambar 3. Kecenderungan peningkatan ini akan terus bertambah, meskipun belum diketahui apakah penambahannya tetap seperti eksponensial ataupun linier.

Ada beberapa penyebab kegagalan perangkat lunak; ketidaktahuan programmer bahwa apa yang dikerjakannya dapat berakibat gagalnya software, kemalasan programmer dalam membuat kode yang bersih, dan adanya proses bisnis tertentu yang menyebabkan

⁴ Confidentiality, Integrity, dan Availability sering disingkat menggunakan huruf depan mereka menjadi CIA. Selain itu ada juga aspek *non-repudiation*, yaitu tidak dapat menyangkal (telah terjadinya transaksi).



```

masscan — sh — 35x9
sh-3.2$ env x='() { :; }; echo vulnerable' sh -c "echo this is a test"
vulnerable
this is a test
sh-3.2$

```

Figure 2: Bug di bash sehingga sisipan perintah dapat dieksekusi

keamanan harus dikorbankan karena berseberangan dengan proses bisnis tersebut. Setidaknya, buku ini akan mencoba membasmi ketidaktahuan.

Fokus Bahasan

Ketika kita mendiskusikan keamanan perangkat lunak, apa maksudnya? Bahasan dari buku ini adalah bagaimana kita memastikan bahwa perangkat lunak yang sudah kita gunakan atau kita kembangkan bebas dari masalah keamanan.

Untuk perangkat lunak yang sudah kita miliki, kita dapat menguji apakah dia memiliki masalah keamanan. Metodologi *penetration testing* sering digunakan untuk menguji hal ini. Pengujian cara ini dapat dikatakan sudah terlambat dan sering membutuhkan biaya yang besar untuk memperbaikinya dibandingkan jika kita sudah mengujinya ketika perangkat lunak tersebut sedang dikembangkan. (Ini akan kita bahas secara lebih rinci.)

Untuk perangkat lunak yang sedang (atau akan) kita kembangkan, maka kita berbicara tentang *secure software development life cycle* (SDLC), dimana pengembangan dapat dibagi menjadi beberapa fase. Intinya adalah isu keamanan sudah menjadi perhatian pada setiap fase pengembangan perangkat lunak.

Perangkat lunak ada yang berbasis *client-server* dan ada yang berbasis web. Kedua jenis perangkat lunak ini membutuhkan cara yang berbeda untuk pengujian keamanannya. Akan ada bahasan yang khusus mengenai masing-masing jenis.

Selain kedua jenis aplikasi tersebut ada juga aplikasi untuk platform *mobile*, seperti yang digunakan pada handphone. Pengujian untuk platform ini juga berbeda dari kedua platform di atas. Ini akan menjadi bahasan terpisah juga.

⁴ Software yang sudah jadi atau software yang dikembangkan sendiri?

⁴ Client-server atau web-based? Keduanya memiliki keuntungan dan kerugian. Pemilihan bergantung kepada jenis aplikasi.

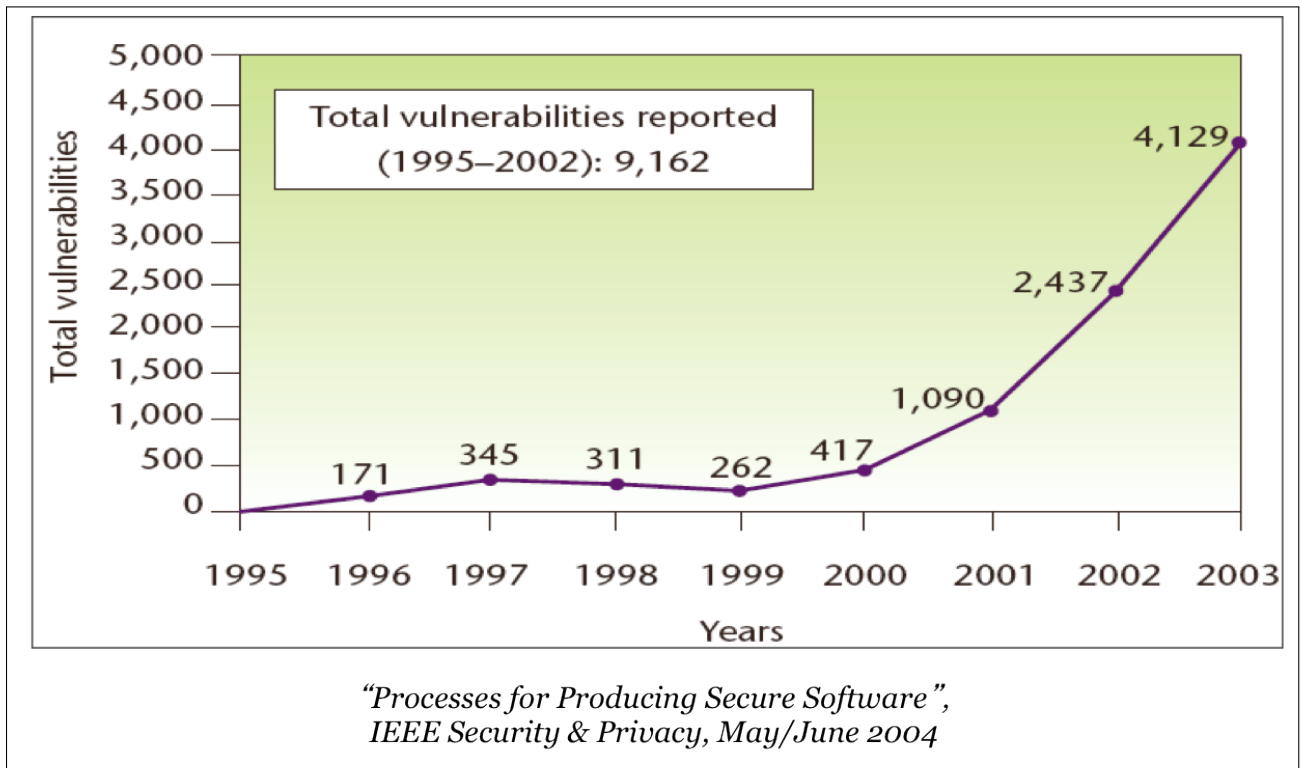


Figure 3: Statistik Meningkatnya Kelemahan Pada Perangkat Lunak

Dasar-Dasar Keamanan

Buku ini membahas tentang keamanan dari perangkat lunak. Untuk itu perlu terlebih dahulu diuraikan mengenai dasar-dasar keamanan secara singkat sehingga cukup untuk melakukan pembahasan mengenai keamanan perangkat lunak. Dasar-dasar keamanan yang lebih rinci dapat dipelajari dengan menggunakan buku referensi lainnya⁵.

Apa yang dimaksud dengan keamanan? Ada tiga faktor utama yang disebut sebagai tujuan (*goals*) atau aspek dari keamanan, yaitu *Confidentiality*, *Integrity*, dan *Availability*. Mereka sering disebut CIA berdasarkan singkatan dari huruf depan mereka. Selain ketiga hal di atas, yang juga dapat disebut sebagai *core security concepts*, ada *general security concepts* lainnya seperti *non-repudiation*, *authentication*, *authorization*, *access control*, dan *auditing*.

Confidentiality

Confidentiality atau kerahasiaan menyatakan bahwa data tidak dapat diakses oleh orang yang tidak berhak. Ketika orang berbicara mengenai keamanan data, faktor ini yang hadir di dalam pikiran kita.

Serangan terhadap aspek ini dilakukan dengan berbagai cara seperti misalnya menyadap jaringan, menerobos akses dari sistem komputer, sampai ke menanamkan *trojan horse* dan *keylogger*⁶ untuk mengambil data secara ilegal. Cara non-teknis dapat dilakukan dengan *social engineering*, yaitu berpura-pura sebagai orang yang berhak mengakses data dan meminta orang lain untuk memberikan data tersebut.

Cara-cara pengamanan terhadap serangan kerahasiaan antara lain adalah dengan menggunakan kriptografi (yaitu mengubah data sehingga terlihat seperti sampah), membatasi akses ke sistem dengan menggunakan *userid* dan *password* sehingga ini dikaitkan juga dengan *access control*.

Salah satu masalah yang terkait dengan aspek kerahasiaan adalah memilah dan melabel data apa saja yang dianggap sebagai data rahasia. Sebagai contoh, apakah data kepegawaian di kantor kita dianggap sebagai rahasia? Apakah data nilai (transkrip) mahasiswa

⁵ Budi Rahardjo. *Keamanan Sistem Informasi Berbasis Internet*. PT Insan Infonesia, 2005

⁶ Keylogger adalah sebuah aplikasi atau alat yang menangkap apa-apa yang kita ketikkan di keyboard.

di kampus merupakan data yang rahasia? Apakah klasifikasi data itu hanya rahasia atau tidak rahasia saja? Atukah ada tingkatannya - seperti *top secret*, rahasia, untuk keperluan internal saja, dan untuk publik? Lantas, siapa yang berhak menentukan tingkat kerahasiaan data ini?⁷ Ini masih merupakan masalah besar di berbagai institusi karena umumnya mereka tidak memiliki panduan atau standar mengenai klasifikasi data.

Integrity

Integrity mengatakan bahwa data tidak boleh berubah tanpa ijin dari pihak yang berhak. Sebagai contoh, data saldo rekening bank milik kita tidak boleh berubah secara tiba-tiba. Transaksi bernilai Rp. 3.000.000,- tidak boleh dapat diubah oleh penyerang menjadi Rp. 3.500.000,-. Atau tujuan transaksi tidak boleh diubah tanpa diketahu. Contoh lain adalah data jumlah pemilih dalam sebuah sistem pemilu atau e-voting tidak boleh berubah tanpa melalui proses yang sah.

Serangan terhadap aspek ini dilakukan melalui serangan *man in the middle* (MITM). Data transaksi ditangkap (*intercepted*) di tengah jalan, dimodifikasi, dan kemudian diteruskan ke tujuan. Penerima tidak sadar bahwa data sudah berubah dan memproses yang sudah berubah ini.

Perlindungan terhadap serangan dapat dilakukan dengan menambahkan *message digest* (*signature*, *checksum*) dalam pesan yang dikirimkan secara terpisah sehingga ketika terjadi perubahan akan terdeteksi di sisi penerima. Banyak aplikasi atau sistem yang belum menerapkan ini sehingga perubahan data yang tidak sah tidak diketahui.

Pencatatan (*logging*) terhadap perubahan data juga harus dilakukan sebagai upaya untuk mengetahui terjadinya serangan terhadap integritas ini. Perlu diingat bahwa pencatatan tidak mencegah terjadinya serangan.

Availability

Aspek *availability* menyatakan bahwa data harus tersedia ketika dibutuhkan. Pada mulanya aspek ini tidak dimasukkan ke dalam aspek keamanan, tetapi ternyata kegagalan berfungsinya sistem dapat mengakibatkan kerugian finansial atau bahkan hilangnya nyawa.

Serangan terhadap aspek ini adalah dengan cara membuat sistem gagal berfungsi, misalnya dengan melakukan permintaan (*request*) yang bertubi-tubi. Serangan ini disebut sebagai *Denial of Service* (DoS). *DoS attack* ini dapat dilakukan pada jaringan dan aplikasi.

⁷ Secara umum seharusnya pemilik aplikasi yang tahu tingkat kerahasiaan data, bukan orang IT. Orang IT dapat dianggap seperti tukang parkir yang menjaga kendaraan di tempat parkir, tetapi dia bukan pemilik dari kendaraan yang diparkir. Juga tukang parkir sesungguhnya tidak tahu nilai dari kendaraan (aset) yang dijaganya.

Serangan yang dilakukan melalui jaringan dapat dilakukan secara terdistribusi, yaitu menggunakan penyerang dalam jumlah yang banyak. Maka muncullah istilah *Distributed DoS* atau *DDoS attack*.

Perlindungan terhadap serangan ini dapat dilakukan dengan menggunakan sistem *redundant* dan *backup*. Keberadaan sistem yang redundan membuat sistem menjadi lebih tahan terhadap serangan. Sebagai contoh, apabila ada satu sistem tidak berfungsi (misal mendapat serangan DoS atau listrik mati), maka sistem lainnya dapat menggantikan fungsinya sehingga layanan tetap dapat diberikan. Permasalahan terhadap sistem yang redundan ini adalah masalah finansial.

Non-repudiation

Aspek *non-repudiation* menyatakan bahwa seseorang tidak dapat menyangkal (telah melakukan sebuah aktifitas tertentu, misalnya telah melakukan transaksi). Aspek ini biasanya dibutuhkan untuk aplikasi yang terkait dengan transaksi.

Serangan atau masalah terhadap aspek ini dapat terjadi jika seseorang menyangkal telah melakukan transaksi. Maka terjadilah *dispute* antar kedua belah pihak akan keabsahan sebuah transaksi. Bagaimana membuktikan bahwa orang yang bersangkutan memang melakukan transaksi?

Salah satu cara untuk mengatasi masalah tersebut adalah dengan melakukan pencatatan (*logging*). Perlindungan terhadap serangan pada aspek ini dapat dikaitkan dengan perlindungan terhadap serangan pada aspek integritas dan ditambahkan dengan pencatatan.

Manajemen Risiko

Masalah keamanan yang terkait dengan sistem teknologi informasi pada awalnya dianggap sebagai masalah teknis. Hal ini menyulitkan komunikasi dengan pihak atasan (manajemen) sehingga penanganan masalah keamanan teknologi informasi tidak mendapat perhatian. Pendekatan yang lazim dilakukan adalah melihat masalah keamanan ini sebagai masalah risiko, sehingga masalah ini dapat dilihat sebagai masalah manajemen risiko (*risk management*).

Ada beberapa metodologi untuk melakukan manajemen risiko. Salah satu panduan yang cukup banyak digunakan adalah metodologi yang didokumentasikan di *NIST SP800-30*⁸. Manajemen risiko terdiri atas tiga hal *risk assessment*, *risk mitigation*, dan *evaluation and assessment*. Manajemen risiko ini menjadi bagian dari setiap langkah pengembangan sistem. Hal ini konsisten dengan usulan-usulan integrasi keamanan dalam pengembangan perangkat lunak.

⁸ Gary Stoneburner, Alice Goguen, and Alexis Feringa. Risk management guide for information technology systems. Technical Report NIST SP 800-30, NIST, 2002

Risk Assessment

Langkah pertama dalam manajemen risiko adalah melakukan *assessment*. NIST SP 800-30 mengusulkan sembilan (9) langkah (kegiatan) dalam melakukan *assessment* ini⁹:

1. System Characterization
2. Threat Identification
3. Vulnerability Identification
4. Control Analysis
5. Likelihood Determination
6. Impact Analysis
7. Risk Determination
8. Control Recommendations
9. Results Documentation

Pada prinsipnya, *risk assessment* bertujuan untuk mengidentifikasi aset yang ingin diproteksi, ancaman, kelemahan, probabilitas kelemahan tersebut dapat dieksploitasi, dan

⁹ Buku ini tidak membahas secara rinci mengenai langkah-langkah ini. Silahkan membaca dokumen tersebut.

Secure Software Development Life Cycle

Pengembangan perangkat lunak secara formal harus mengikuti langkah atau tahapan tertentu, yang dikenal dengan nama *Software Development Life Cycle* (SDLC). Pengembangan diawali dengan tahap *requirement* untuk kemudian dilanjutkan dengan desain, pengembangan *test plan*, implementasi (*coding*), pengujian, dan peluncuran (*deployment*). Secara rinci, SDLC dapat dipelajari dari Pressman.

Hal yang belum nampak secara eksplisit pada *conventional* SDLC adalah aspek keamanan. Keamanan seharusnya hadir pada setiap tahapan SDLC. Ada beberapa pendekatan tentang *secure* SDLC. Gary McGraw menjabarkan pengembangan perangkat lunak yang aman dalam bukunya ¹⁰. Selain itu ada beberapa metodologi lain yang akan dibahas lebih lanjut.

¹⁰ Gary McGraw. *Software Security: Building Security In*. Addison-Wesley, 2006

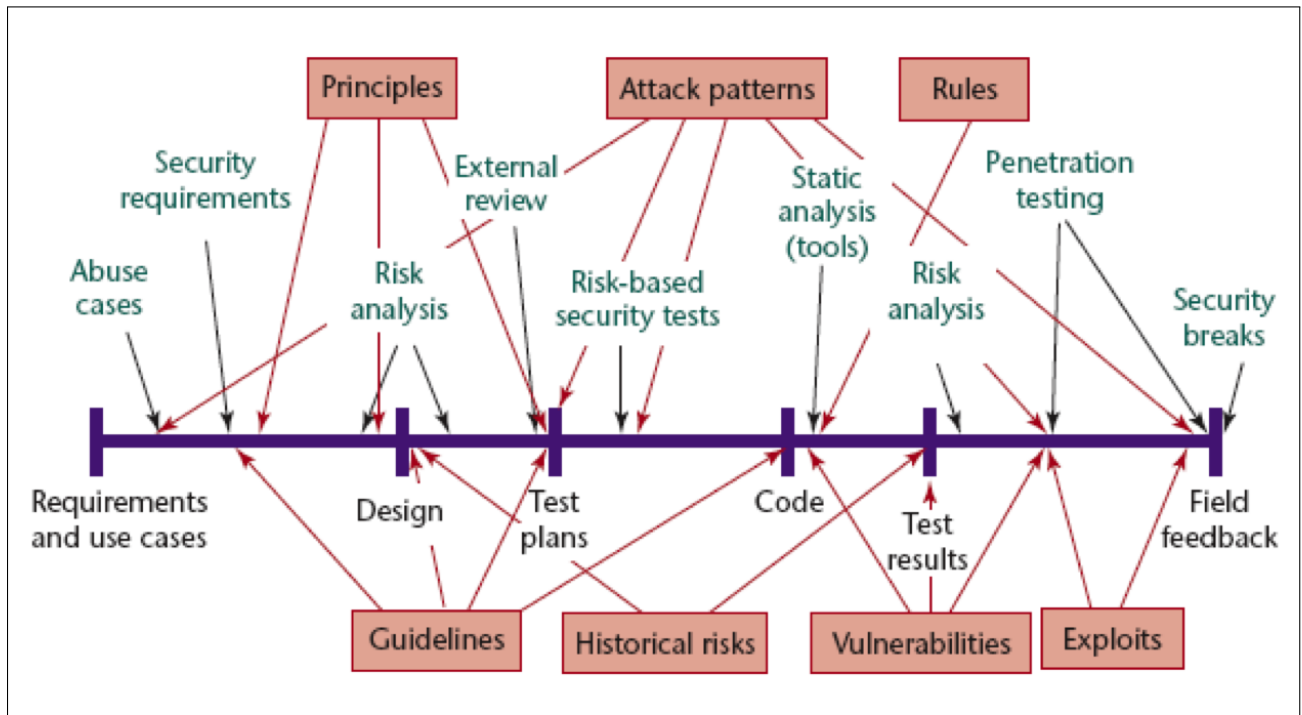


Figure 4: Secure SDCL menurut Gary McGraw

Seperti dapat dilihat pada gambar 4, keamanan - bagian yang berwarna hijau - dipertimbangkan pada setiap tahapan SDLC. *Conventional SDLC* ditunjukkan dengan warna hitam.

Security Requirement

Dalam pengembangan perangkat lunak, *requirement* merupakan sebuah hal yang sangat esensial. Banyak pengembangan perangkat lunak yang gagal dikarenakan *requirement* yang berubah-ubah. Waktu pengembangan bertambah panjang sesuai dengan perubahan *requirement*.

Pengembangan perangkat lunak tanpa *security requirement* mengalami masalah tambahan, yaitu masalah yang terkait dengan keamanan. Namun penambahan *security requirement*, yang merupakan *non-functional requirement* malah dianggap sebagai tambahan beban sehingga tidak dilakukan. Padahal tanpa adanya *security requirement*, masalah yang timbul di kemudian hari justru akan mahal untuk diperbaiki.

Security requirement dikaitkan dengan faktor yang terkait dengan keamanan, yaitu *confidentiality*, *integrity*, dan *availability*. Selain komponen di atas, ada juga faktor lain yang diusulkan oleh Mano¹¹ seperti *authentication requirement*, *authorization requirement*, *auditing requirement*, *session management requirement*, *errors and exceptions management requirement*, *configuration parameters management requirement*, *sequencing and timing requirement*, *archiving requirement*, *international requirement*, *deployment requirement*, *procurement requirement*, *antipiracy requirement*.

Contoh *security requirement* antara lain:

- Password harus dilindungi ketika diketikkan / ditampilkan harus di"masking" (misal dengan menggunakan karakter bintang). Ketika disimpan, password harus disimpan dalam bentuk *hashed*.
- Apakah password memiliki usia (password aging)? Misalnya, apakah password harus diganti setiap bulan? Jika harus diganti, apakah password yang baru boleh sama dengan password sebelumnya.
- Bolehkah pengguna mengakses layanan dari dua tempat yang berbeda pada saat yang bersamaan? Layanan transaksi, seperti misalnya internet banking, biasanya tidak memperkenankan kondisi ini tetapi ada layanan yang memperbolehkan *multiple concurrent access*.
- Perubahan data harga harus tercatat dengan menyertakan *timestamp* dan identitas pengguna yang melakukan perubahan.

¹¹ Mano Paul. *Official (ISC) Guide to the CSSLP*. CRC Press, 2011

- Waktu ketika aplikasi dijalankan (*start*) dan dimatikan (*shutdown*) harus dicatat (*logged*).

Sumber dari *security requirement* dapat diperoleh dari *internal* dan *external*. Dari *internal* sumber dapat berasal dari kebijakan, standar, *guidelines*, dan kebiasaan. Dari sumber *external* sumber dapat berasal dari regulasi, *compliance*, dan seterusnya.

Notasi apa yang baik untuk mendokumentasikan *security requirement*? Saat buku ini ditulis, belum ada sebuah standar baku yang digunakan secara umum untuk menuliskan *security requirement*. Kebanyakan masih menggunakan cara deskriptif, yaitu dengan menggunakan kata-kata penjelasan. Di kemudian hari mungkin akan disepakati adanya notasi yang lebih mudah digunakan untuk *security requirement* ini. Ini mirip dengan *functional requirement*, yang mana pada awalnya menggunakan deskriptif tetapi kemudian mulai digunakan notasi yang lebih “formal” seperti UML.

Salah satu standar *security requirement* yang sering digunakan adalah *Common Criteria for Information Technology Security Evaluation* (atau lebih sering disebut Common Criteria saja). Standar ini juga terkait dengan ISO 15408 tentang *Evaluation Criteria for IT Security*.

Latihan

Sebuah instansi pemerintah ingin membuat sebuah portal untuk melayani publik dan internal instansi. Portal ini menyediakan layanan perijinan dan juga menyediakan data statistik. Ada pengguna tamu (*guest*) yang hanya dapat melihat-lihat, ada pengguna yang terdaftar, dan ada penyedia informasi. Buat *security requirement* untuk portal ini.

Security Requirement

Pengembangan perangkat lunak (atau aplikasi) dimulai dengan sebuah *requirement*, yang berisi daftar fungsi-fungsi yang akan diberikan oleh perangkat lunak tersebut. Tanpa ada *requirement* pengembang (developer) akan kesulitan membuat produk yang diinginkan. Bahkan seringkali terjadi perdebatan (keributan) antara pengembang dan pemilik aplikasi karena ketidakjelasan (ketiadaan) *requirement* ini. Dikatakan bahwa banyak aplikasi yang gagal (dalam artian terlambat dikembangkan, biaya yang membengkak, dan bahkan tidak selesai) gara-gara ketidakjelasan *requirement* ini.

Menurut ISO 24765¹², definisi dari *requirement* adalah sebagai berikut:

A condition or capability needed by a user to solve a problem or achieve an objective.

A condition or capability that must be met or possessed by a system ... to satisfy a contract, standard, specification, or other formally imposed document.

Singkatnya *functional requirement* berisi fungsi-fungsi yang harus dipenuhi oleh aplikasi. Di sana dijelaskan hubungan antara *input*, *output*, dan hubungan antara keduanya.

Selain *functional requirement*, ada juga *non-functional requirement*. Yang ini berhubungan dengan *performance* (kinerja) dan *security*.

Sisi keamanan dari pengembangan perangkat lunak juga membutuhkan *security requirement*, yang berisi hal-hal terkait keamanan yang harus ada dalam aplikasi. Sayangnya *security requirement* belum dipahami sehingga masih belum banyak digunakan. Padahal - sama seperti *functional requirement* - keberadaan *security requirement* sangat esensial.

Permasalahan *security requirement* adalah dia sering kali dianggap menambah beban pengembang (dengan penambahan *requirement* tersebut) sehingga menambah waktu pengembangan dan juga akhirnya menambah biaya. Padahal ini tidak perlu terjadi jika pengembang mengerti hal ini.

Untuk memahami apa itu *security requirement*, berikut ini beberapa contoh. Diskusi yang lebih rinci akan dibahas kemudian.

¹² ISO. ISO/IEC/IEEE 24765:2010: Systems and software engineering – vocabulary, 2010

1. Dalam sebuah aplikasi berbasis web. Apakah pengguna harus terdaftar? Apakah pengguna *anonymous* boleh mengakses layanan?
2. Password. Berapa panjang minimal karakternya? Apakah ada panjang maksimum? Apakah password harus mengandung karakter tertentu? Apakah ada *password aging*? Jika ya, berapa lama password harus diganti? Bolehkah menggunakan password lama? (Apakah ada history?)
3. Apakah *concurrent access* diperbolehkan? (Akses ke satu akun dari lebih dari dua sesi dari tempat yang berbeda pada saat bersamaan.)

Security requirement dibuat mengacu kepada aspek keamanan, yaitu CIA (Confidentiality, Integrity, Availability). Selain ketiga aspek tersebut, *security requirement* dapat juga ditambahkan aspek lain. Sayangnya saat penulisan ini belum ada sebuah standar atau *checklist* yang dapat membantu membuat daftar *security requirement* yang komplit. *Security requirement* ini juga nantinya bergantung kepada bidang industri yang akan digunakan.

Confidentiality Requirement

Pada prinsipnya, *confidentiality requirement* menampilkan daftar apa-apa saja yang terkait dengan kerahasiaan. Berikut ini beberapa contohnya.

1. Data sensitif dalam perjalanan, dalam proses, dan ketika disimpan harus dalam format yang terenkripsi. Sebagai contoh, data ketika disimpan dalam database (atau berkas biasa) tidak boleh dalam bentuk *plain text*. Data sensitif yang dikirimkan melalui jaringan juga harus terenkripsi. (Persyaratan ini nantinya membuat aplikasi berbasis web tidak boleh menggunakan protokol HTTP dan harus menggunakan HTTPS.)
2. Password ketika ditampilkan di layar, tidak boleh dalam bentuk yang dapat terbaca. *Password must be masked*.
3. Password tidak boleh dicatatkan pada log.
4. Pengguna tidak dapat melihat transaksi (data) dari pengguna lainnya.

Integrity Requirements

1. Semua transaksi harus memiliki *checksum* dalam bentuk *hashed*. Data hash harus dikirimkan secara terpisah.

Availability Requirements

Persyaratan terkait dengan aspek *availability* biasanya terkait dengan tingkat kepentingan (kritikalitas) dari aplikasi. Ada aplikasi yang menentukan hidup-matinya perusahaan, tetapi ada juga aplikasi yang hanya menyebabkan ketidaknyamanan ketika tidak dapat diakses.

1. Aplikasi harus tersedia pada jam kerja.

Authentication Requirements

Beberapa contoh *authentication requirement*:

1. Pengguna dari aplikasi harus diketahui (authenticated). Tidak boleh ada pengguna anonim (*anonymous*).
2. Authentication harus dilakukan dengan menggunakan dua faktor.

Authorization Requirements

Persyaratan ini biasanya terkait dengan *roles* dari pengguna. Untuk itu perlu ada definisi dahulu tentang siapa-siapa pengguna aplikasi ini dan kebijakan terkait dengan pengguna tersebut.

Audit Requirements

1. Semua data transaksi harus dicatat (logged) di dua tempat.

Session Management Requirements

Errors and Exception Management Requirements

Configuration Management Requirements

Archiving Requirements

Deployment Requirements

Notasi Security Requirements

Setelah mengetahui apa-apa yang ingin kita persyaratkan dalam *security requirements*, pertanyaannya adalah bagaimana menuliskan *requirements* tersebut? Apakah ada notasi khusus?

Pada saat buku ini ditulis, belum ada notasi khusus untuk penulisan *security requirement*. Panduan dasar yang digunakan adalah notasi

yang digunakan sama dengan yang digunakan dalam *functional requirement*. Selain itu perlu diperhatikan juga notasi yang akan digunakan oleh pengembang. Perlu diingat bahwa prinsip utamanya adalah dokumen ini digunakan untuk berdialog antara pemilik aplikasi dan pengembang, sehingga semuanya harus dapat berdiskusi tanpa salah pengertian.

Abuse Case

Sebetulnya *abuse case* berada di luar lingkup dari *security requirements*, tetapi karena dia terkait maka akan dijabarkan di sini. Jika *security requirements* menjabarkan apa-apa saja yang kita inginkan (dari kaca mata security) ada dalam aplikasi, maka *abuse case* menjabarkan apa-apa saja yang mungkin dapat terjadi sehingga perlu dihindari.

Latihan

Sebuah bank ingin membuat layanan *internet banking*. Buat *confidentiality requirement* dari layanan tersebut.

Secure Design

Setelah *requirement* tersedia, pengembang dapat mulai masuk ke tahap desain. Dari kacamata fungsional, desain menghasilkan arsitektur dari software yang dikembangkan. Masalah desain terkait dengan *business logic flaw*, bukan *bugs* karena kode belum dibuat. Bahkan jika desain sudah salah, maka kode yang dibuat dengan menggunakan bahasa pemrograman apapun akan tetapi menghasilkan lubang keamanan.

Ini analoginya adalah kalau kita mendesain sebuah perkantoran dengan tanpa pintu. Dinding mau diimplementasikan dengan bata ataupun tripleks tidak berpengaruh karena desain memang tidak memperhatikan masalah keamanan.

Masalah keamanan ini harus ditemukan dengan segera karena memperbaiki masalah *security* setelah software tersebut menjadi bagian dari *production* biayanya dapat membengkak seratus (100) kali lebih mahal.

Dari kacamata keamanan, *secure design* mengembangkan kendali atas *security requirement* yang telah dikembangkan pada fasa sebelumnya. Sebagai contoh, *security requirement* menyatakan bahwa pengguna tidak boleh mengakses (login) dari dua tempat yang berbeda pada saat yang sama. Dengan kata lain, *concurrent login* tidak diperkenankan. Kendali dapat dilakukan dengan menggunakan kombinasi *cookies*, nomor IP, jenis browser, dan informasi lainnya ¹³.

Contoh kendali lain adalah dalam hal *password recovery*. Bagaimana memastikan bahwa pengguna yang sah membutuhkan password baru? Salah satu cara kendali adalah dengan menanyakan pengguna dengan pertanyaan tertentu seperti misalnya warna favorit dari pengguna ¹⁴. Pertanyaan lain yang juga sering digunakan adalah nama binatang peliharaan pengguna. Namun ini juga memiliki kelemahan karena seringkali pengguna menampilkan data dan foto binatang peliharaannya di akun media sosialnya. Tentunya menanyakan tanggal kelahiran pengguna terlalu mudah untuk diketahui sehingga masih belum cukup sebagai kendali pengamanan.

Bagaimana notasi dari *security design*? Bentuk notasi dari *security*

¹³ Dahulu, kendali dilakukan hanya dengan menggunakan *cookies* saja, tetapi ternyata ini tidak cukup. Orang dapat mencuri *cookies* tersebut, memasangnya di komputer lain, dan mengakses layanan secara bersamaan.

¹⁴ Menanyakan nama warna memiliki kelemahan karena ada beberapa nama yang menjadi favorit seperti misalnya merah, biru, putih, hitam, dan sejenisnya. Penyerang dapat mencoba-coba dengan warna tersebut dahulu sebelum memilih nama warna yang agak jarang digunakan.

design sama dengan notasi desain. Jika desain fungsional dilakukan dalam bentuk flow chart, DFD, atau UML, maka desain dari kendali menggunakan notasi yang sama.

Security umumnya dijabarkan sebagai confidentiality, integrity, availability, dan lainnya. *Security design* harus memperhatikan aspek itu juga. Sebagai contoh, untuk memenuhi *requirement* terhadap aspek *confidentiality* secara umum kriptografi dapat digunakan. Untuk memenuhi aspek *integrity* dapat digunakan *hash function*. Untuk memenuhi aspek *availability* dapat digunakan *data replication*. Dan seterusnya.

Mano ¹⁵ dan Seacord ¹⁶ menampilkan beberapa prinsip desain yang dapat digunakan agar keamanan dapat tercapai. Prinsip-prinsip ini awalnya dikembangkan oleh ¹⁷ di tahun 1975 dan masih cocok untuk digunakan saat ini.

- **Economy of mechanism.** Usahakan desain sesederhana dan sekecil mungkin.
- **Fail-safe defaults.** Ketika program gagal berfungsi, maka sistem berada pada kondisi yang aman. Akses menggunakan prinsip *permission* bukan *exclusion*.
- **Complete mediation.** Segala usaha untuk mengakses sebuah obyek harus dicek otoritasnya.
- **Open design.** Desain tidak boleh dirahasiakan. Kekuatan dari keamanannya bukan pada kerahasiaan dari desainnya.
- **Separation of privilege.** Pemisahan kunci, menjadi dua dan harus digunakan bersama misalnya, dapat meningkatkan keamanan.
- **Least privilege.** Jika aplikasi tidak perlu menggunakan otoritas tingkat admin, maka gunakan tingkat pengguna biasa saja.
- **Least common mechanism.** Mekanisme yang sama untuk berbagai pengguna yang level otoritasnya berbeda dibuat seminimal mungkin.
- **Psychological acceptability.** Solusi keamanan harus mudah digunakan oleh pengguna sehingga selalu diterapkan oleh pengguna dan tidak dilanggar.

¹⁵ Mano Paul. *Official (ISC) Guide to the CSSLP*. CRC Press, 2011

¹⁶ Robert C. Seacord. *Secure Coding in C and C++*. Addison-Wesley, 2005

¹⁷ Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, September 1975

Secure Coding

Desain dari software diimplementasikan dalam bentuk kode. Meskipun kita sudah memiliki *security design* sebagai tambahan dari desain secara fungsional, keamanan dari software masih bergantung kepada bagaimana desain tersebut diimplementasikan. Kecerobohan pemrogram (*programmer, coder*) dapat menghasilkan lubang keamanan yang seharusnya tidak ada.

Ada bahasa pemrograman yang membatasi gerak pemrogram sehingga dapat mengurangi kesalahan yang mungkin terjadi. Sebagai contoh, bahasa pemrograman yang memiliki sifat *strongly typed* (seperti bahasa Java) mengurangi kemungkinan terjadinya kesalahan tipe data. Namun, kesalahan masih dapat terjadi dari sisi lainnya.

Di sisi lain ada bahasa pemrograman yang sangat *powerful* dan sangat memberikan keleluasaan pemrogram untuk menggunakannya (*abuse?*) sehingga mudah bagi pemrogram untuk berbuat kesalahan. Bahasa C merupakan salah satu contohnya¹⁸. Untuk itu ada beberapa hal yang dapat diajarkan agar kode yang dihasilkan masih memenuhi aspek keamanan.

¹⁸ Robert C. Seacord. *Secure Coding in C and C++*. Addison-Wesley, 2005

Buffer Overflow

Salah satu kesalahan yang paling sering terjadi pada pemrograman adalah *buffer overflow*. Pada prinsipnya, kesalahan ini terjadi karena kita menyediakan memori (buffer) yang tidak cukup untuk sebuah variabel. Sebagai contoh, berapa besar memori yang kita alokasikan untuk variabel *userid*? Cukupkah kita mengalokasikan empat puluh (40) karakter untuk variabel ini? (Jarang orang memiliki *userid* yang lebih panjang dari 40 karakter.) Bagaimana jika pengguna memasukkan lebih dari 40 karakter? Buffer tersebut dapat dilanggar dengan berbagai konsekuensi akibat.

Pengujian

Pengujian (*testing*) dilakukan untuk memastikan bahwa perangkat lunak atau sistem yang kita kembangkan sudah memenuhi apa yang kita inginkan (kebutuhan). Pengujian adalah membandingkan antara *requirement* dan implementasi. Skenario pengujian dan datanya harus disiapkan sebelum kode dibuat. Jika tidak, kode akan kita uji dengan apa? Pada bagian ini akan dibahas pengujian dengan lebih rinci.

Pembuktian versus Pengujian

Sebetulnya untuk mengetahui apakah sebuah sistem sudah benar atau tidak dapat kita lakukan dengan dua cara, yaitu melalui pembuktian (*proof*) dan melalui pengujian (*testing*). Perbedaan antara keduanya dapat dijelaskan dengan contoh berikut.

Berapa gaya tarik bumi (*gravity*)? Ada dua cara untuk mencari gaya tarik bumi. Cara pertama adalah dengan menurunkan dari berbagai persamaan sehingga akhirnya diperoleh nilai $9,8 \text{ m/detik}^2$. (Ada yang tahu persamaan-persamaan apa saja yang digunakan untuk menurunkan angka ini?)

Cara kedua adalah dengan melakukan pengukuran (pengujian) melalui percobaan dengan menjatuhkan benda dengan variasi berat dengan ketinggian tertentu dan mengukur waktunya sampai benda tersebut *hit the ground*. Dari berbagai pengukuran akan diperoleh nilai yang sama.

Pembuktian memberikan nilai kepercayaan yang lebih tinggi daripada pengujian (pengukuran), akan tetapi seringkali pembuktian sulit untuk dilakukan. Seringkali sistem yang ingin dibuktikan memiliki tingkat kompleksitas yang tinggi sehingga sulit untuk dilakukan pembuktian. Perangkat lunak masuk kategori ini. Sehingga pembuktian hanya dapat dilakukan untuk sistem yang ukurannya kecil, atau sering disebut sebagai *toy problems*.

Ilmu yang membahas tentang pembuktian ini dikenal sebagai *formal methods*, metoda formal. Bidang ini masih kental aspek penelitiannya karena tingginya kompleksitas yang harus dihadapi. Terkait dengan hal ini adalah *automated theorem prover*. Untuk bidang

perangkat keras (*hardware*)¹⁹, metoda formal lebih banyak keberhasilannya dibandingkan dengan di bidang perangkat lunak.

Buku ini akan lebih fokus kepada pengujian.

¹⁹ Budi Rahardjo. *Formal Verification of Asynchronous Systems*. PhD thesis, University of Manitoba, 1996

Black Box versus White Box

Pengujian dapat dilakukan dengan menganggap aplikasi sebagai sebuah sistem yang tertutup, yang hanya dapat diakses melalui *input* dan *output*. Pendekatan ini disebut *black box*.



Figure 5: Pengujian secara blackbox

Pengujian secara *black box* merupakan pendekatan yang paling banyak digunakan. Hal ini disebabkan oleh beberapa hal, seperti yang dapat dilihat pada daftar berikut ini.

- **Tidak memiliki akses ke kode sumber.** Seringkali kita tidak memiliki kode sumber (*source code*) dari aplikasi yang akan diuji. Jika kita membeli aplikasi yang sudah jadi, maka ini yang sering terjadi. Kita hanya memiliki berkas *executable*.
- **Simulasi pengguna.** Secara umum, kondisi ini - aplikasi hanya dapat akses ke *input* dan *output* - merupakan kondisi yang paling umum. Pengguna hanya dapat mengakses *input* dan *output*. Pengujian mensimulasikan kondisi ini.
- **Lebih mudah diotomatisasi.** Ada beberapa tools atau framework yang dapat digunakan untuk melakukan pengujian secara otomatis, yaitu dengan menghasilkan berbagai *input* secara random atau mengikuti sebuah pola tertentu dan memantau *output*nya. *Fuzzing tools*, yang akan dibahas secara khusus, dapat digunakan untuk mengotomatisasi ini.

Pengujian *black box* digunakan untuk simulasi penyerang yang tidak memiliki akses ke sistem kita (*unauthorized user*).

Pengujian *white box* dilakukan dengan asumsi penguji memiliki akses kepada kode sumber dan data detail lainnya (seperti topologi jaringan, nomor IP, akun yang legal, dan konfigurasi dari sistem lainnya). Pengujian *white box* semestinya dapat memberikan hasil yang lebih banyak dari sekedar *input* dan *output* saja. Pengujian *white box* mensimulasikan akses oleh pengelola atau pengembang (*developer*) sistem.

Pengujian *white box* seharusnya menemukan kelemahan lebih banyak dari *black box*, namun lebih sulit dilakukan. *State of the art* dari *white box* masih melibatkan kegiatan manusia secara ekstensif.

Selain kedua pendekatan di atas, ada juga pengujian yang disebut *grey box*. Pengujian cara ini adalah merupakan antara cara *black box* dan *white box*. Pada pengujian *grey box*, penguji diberi akses kepada sistem dengan otoritas pengguna biasa. Misalnya, pada sebuah aplikasi internet banking, penguji diberi akun dan password untuk mengakses akun internet bankingnya.

Manual Atau Otomatis

Pengujian dapat dilakukan secara manual atau otomatis. Umumnya pengembang melakukan pengujian secara manual karena tidak tahu bahwa ada mekanisme untuk melakukan pengujian secara otomatis. Pengujian secara manual digunakan untuk hal-hal yang sederhana dan tidak memerlukan pengulangan, tetapi pada kenyataannya perangkat lunak harus diuji berkali-kali. Jarang sekali kita membuat sebuah perangkat lunak yang langsung berjalan dengan benar dalam satu kali percobaan. Sehingga, sesungguhnya pengujian secara otomatis merupakan hal yang harus diketahui.

Misalnya kita membuat modul "login" untuk sebuah aplikasi. Kita menguji modul tersebut secara terpisah dari modul lainnya. Pengujian katakanlah dilakukan dengan 100 kali (data) percobaan. Apabila ada perubahan, maka modul tersebut harus kita uji lagi. Maka kita harus melakukan 100 kali pengujian lagi. Katakanlah ada perubahan versi, maka dia harus diuji 100 kali lagi. Jika ini dilakukan secara manual, dapat dibayangkan betapa repotnya.

Ada beberapa tools (framework) yang dapat digunakan untuk melakukan pengujian secara otomatis ini. Biasanya tools ini bergantung kepada bahasa pemrograman yang digunakan. Sebagai contoh, untuk yang menggunakan bahasa pemrograman Java dapat menggunakan **JUnit**²⁰. Untuk menguji aplikasi yang berbasis web dapat digunakan Selenium²¹.

²⁰ Informasi mengenai JUnit dapat dilihat di junit.org. JUnit is a simple framework to write repeatable tests.

²¹ Informasi mengenai Selenium dapat diperoleh di <http://docs.seleniumhq.org/>

Skenario Pengujian

Pengujian dilakukan dengan menggunakan skenario atau kasus-kasus (*test cases*). Kasus-kasus ini seharusnya tersedia sebelum kode dibuat sehingga pengembang dapat memastikan kodenya benar atau masih salah dengan menggunakan kasus-kasus tersebut.

Ada beberapa kasus yang harus ditangani:

- normal;

- ekstrim;
- abuse.

Masing-masing kasus ini akan diuraikan pada bagian selanjutnya.

Kasus Normal

Untuk menjelaskan hal tersebut, mari kita ambil contoh. Kita diminta untuk membuat sebuah modul atau fungsi untuk mengurutkan data yang berupa bilangan *integer* positif dari kecil ke besar. Contoh data untuk kasus normal adalah sebagai berikut:

input: 1, 7, 5, 3, 2, 6, 4

output: 1, 2, 3, 4, 5, 6, 7

Apakah dengan satu kali pengujian tersebut kita yakin bahwa modul kita sudah benar? Seharusnya kita membutuhkan beberapa kali pengujian²². Kita dapat membuat beberapa data lagi untuk kasus normal.

- 1, 7, 3, 2, 5, 6, 4
- 2, 1, 7, 3, 4, 5, 6
- 3, 9, 2, 11, 8, 34
- ... dan seterusnya.

Dalam contoh di atas kita masih harus membuat daftar hasil (keluarannya) benarnya. Sebagai contoh untuk data ke tiga, keluarannya harus 2, 3, 8, 9, 11, 34. Cara yang lebih baik adalah dengan menggunakan tabel.

Pembuatan data di atas dapat kita lakukan secara manual atau dihasilkan (*generate*) oleh program secara otomatis.

Pengujian yang berkali-kali ini disebut *regression*(?). Apakah dengan menguji berkali-kali tersebut kita yakin seratus persen bahwa modul tersebut telah kita buat dengan benar? Tidak. Untuk memastikan bahwa modul sudah benar, secara teori dia harus diuji dengan semua kombinasi yang memungkinkan. *All possible combinations*.

Jika kita membuat sebuah *adder* (penjumlah) untuk dua bilangan 4-bit, maka akan ada 2^8 kombinasi. Jumlah 256 kombinasi ini masih dapat dilakukan secara *exhaustive*, yaitu mencoba semua kombinasi. Begitu jumlah bit-nya meningkat, maka jumlah kombinasinya juga meningkat secara eksponensial. Sebagai contoh, *long integer* di dalam bahasa C didefinisikan sebagai 32-bit. Untuk modul *adder* dengan *long integer* dibutuhkan 2^{64} atau 18446744073709551616 ($1,8446744073709551616 \times 10^{19}$) kombinasi. Menguji semua kombinasi ini tidak dapat dilakukan dengan ketersediaan komputasi saat ini²³.

²² Berapa kali seharusnya pengujian dilakukan menurut statistik?

²³ Bayangkan jika satu kombinasi membutuhkan waktu 1 milidetik, berapa waktu yang dibutuhkan untuk menguji semua kombinasi?

Untuk kasus modul *sorting* kita tidak dimungkinkan untuk menguji semua kombinasi. Bayangkan, ada berapa permutasi bilangan *integer* yang perlu diurut tersebut? Maka pengujian terpaksa menggunakan sampel.

Kasus Ekstrim

Salah satu cara untuk meningkatkan kepercayaan adalah dengan menguji modul dengan data yang mewakili kasus ekstrim. Dalam contoh modul pengurutan sebelumnya, kasus ekstrim dapat ditampilkan sebagai berikut.

- 1, 2, 3, 4, 5, 6, 7 (data sudah terurut);
- 7, 6, 5, 4, 3, 2, 1 (data sudah terurut secara terbalik);
- 1, 1, 1, 1, 1, 1, 1 (data sama semua);
- 0, 0, 0, 0, 0, 0, 0 (data minimal);
- 65535, 65535, 65535, 65535, 65535 (data maksimal);
- 1,8,2,9,11,37,8,19,29,4,5,91,... (jumlah data yang banyak).

Kita dapat menambahkan data ekstrim lainnya. Sayangnya kelengkapan data ini bergantung kepada kemampuan kita untuk melengkapinya.

Kasus Abuse

Kalau dalam kasus-kasus sebelumnya kita memberikan data yang benar dan fokus kepada fungsionalitas sesuai dengan yang diinginkan, maka pada kasus ini kita mulai memikirkan aspek keamanannya. Kita mulai memikirkan kasus atau data yang seharusnya tidak boleh diberikan ke modul tersebut.

- 7, -3, 1, 2, 4, 5 (data negatif);
- 7, 3, a, 8, 5, 2 (ada data yang bukan *integer*, bukan tipe data yang legal);
- 7, 3, 1.2, 7, 5, 2 (data bilangan riil);
- 7, 2382738972837892738, 1, 2 (bilangan terlalu besar);
- 7, 3, , , 2 (ada data yang kosong).

Kesulitan yang ada dalam membuat daftar kasus abuse adalah kita tidak dapat membuat daftar *all possible abuse*. Data yang ada adalah data yang teringat.

Standard Data Set

Seringkali ada pihak lain yang sudah pernah mengembangkan modul yang sama atau mirip. Mereka menerbitkan juga data yang digunakan untuk menguji modul mereka. Jika banyak orang yang mengembangkan hal yang sama, seperti misalnya di dunia penelitian, maka akhirnya akan ada kumpulan data standar yang digunakan untuk pengujian. Ini sering juga dikenal dengan *standard corpus*.

Sebagai contoh, jika kita mengembangkan aplikasi yang terkait dengan pemrosesan citra (*image processing*) maka ada data standar untuk pengujian. Standar data ini berisi gambar *Leena*, *Baboon*, *Bridge*, dan seterusnya. Coba cari apakah aplikasi atau modul Anda juga memiliki standar data pengujian.

Fuzzing

Kasus abuse dapat kita perdalam lebih lanjut lagi dengan menggunakan cara yang lebih otomatis, yaitu dengan menggunakan teknik *fuzzing*²⁴. Pada prinsipnya teknik *fuzzing* adalah mencoba memberi data invalid secara otomatis atau semi otomatis. Tujuannya adalah untuk mencari kesalahan yang misalnya dapat membuat sistem menjadi gagal berfungsi (*crash*). Dari kacamata *security* harapannya kegagalan fungsi ini menimbulkan celah keamanan yang dapat dieksploitasi.

Ada beberapa tools atau framework dari *fuzzing* ini. Mereka umumnya masih dalam tahap penelitian.

²⁴ Ari Takanen, Jared DeMoot, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008

Penutup

Keamanan dari perangkat lunak merupakan sebuah bidang yang masih “baru”, dilihat dari segi usia. Namun masalah keamanannya adalah hal yang riil. Kita sudah terlalu banyak bergantung kepada perangkat lunak. Untuk itu ilmu mengenai hal ini harus digali dan disebar. Sayangnya buku referensi mengenai hal ini masih belum banyak. Itulah alasan menerbitkan buku ini.

Salah satu masalah membuat buku dari ilmu yang masih baru adalah ada banyak hal yang masih berkembang. Meskipun prinsip-prinsip yang digunakan pada keamanan - seperti *Confidentiality*, *Integrity* dan *Availability* - tidak terlalu banyak berubah, *tools* yang digunakan sangat cepat berubah. Ada kemungkinan buku ini menjadi cepat kadaluwarsa. Pada saat itu, buku ini harus saya perbaiki lagi. Mudah-mudahan hal ini tidak terlalu sering terjadi.

Bibliography

- [1] ISO. ISO/IEC/IEEE 24765:2010: Systems and software engineering – vocabulary, 2010.
- [2] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley, 2006.
- [3] Richard Miller. Social network analysis. *IEEE Transaction of Networks*, 2013.
- [4] Mano Paul. *Official (ISC) Guide to the CSSLP*. CRC Press, 2011.
- [5] Ivars Peterson. *Fatal Defect: Chasing Killer Computer Bugs*. Random House, 1995.
- [6] Budi Rahardjo. *Formal Verification of Asynchronous Systems*. PhD thesis, University of Manitoba, 1996.
- [7] Budi Rahardjo. *Keamanan Sistem Informasi Berbasis Internet*. PT Insan Infonesia, 2005.
- [8] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, September 1975.
- [9] Robert C. Seacord. *Secure Coding in C and C++*. Addison-Wesley, 2005.
- [10] Gary Stoneburner, Alice Goguen, and Alexis Feringa. Risk management guide for information technology systems. Technical Report NIST SP 800-30, NIST, 2002.
- [11] Ari Takanen, Jared DeMoot, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

Mengenai Penulis

Penulis merupakan staf pengajar dari Sekolah Teknik Elektro dan Informatika (STEI), Institut Teknologi Bandung. Penulis mulai belajar pemrograman secara resmi di tahun 1981 ketika mendapatkan komputer Apple. Sejak saat itu pemrograman merupakan hobynya. Bahasa pemrograman yang dikuasainya antara lain assembly (6502, 6800-series), BASIC, FORTRAN, Pascal, Perl, Promela, VHDL, dan masih banyak lainnya.