The quest for secure code

Paolo Perego
Owasp Italy

Security consultant –
Spike Reply
thesp0nge@gmail.com

OWASP
@eAcademy 2006

4-7 Ottobre 2006

The OWASP Foundation
http://www.owasp.org

Agenda

- Introduzione
- Cosa devo difendere
  - Application security
- Come lo difendo
  - Strategie di safe coding
  - Code review
- Con cosa lo difendo
  - Orizon
- Conclusioni

Introduzione

- All'interno del ciclo di vita del software non ci sono fasi specifiche per la sicurezza
- Un errore in fase di design comporta un effort crescente se si interviene in fasi successive del ciclo di vita
- Il team di sviluppo deve essere supportato durante tutte le fasi della scrittura del codice:
  - Prima
  - Durante
  - Dopo

Introduzione

- Specializzare le figure e i team all'interno del ciclo di vita di un'applicazione
- Introdurre nuove attività a supportare le fasi classiche del ciclo di vita del software
- Porre il codice in primo piano
  - Un errore a livello applicativo è un rischio
  - Comprare un firewall non mi aiuterà
- Avere solide fondamenta
  - Investire nelle fase di analisi e di progettazione

Agenda

- Introduzione
- Cosa devo difendere
  - Application security
- Come lo difendo
  - Strategie di safe coding
  - Code review
- Con cosa lo difendo
  - Orizon
- Conclusioni

Cosa devo difendere: Application security

- Application security != Network security
  - Un firewall non serve a proteggere le mie applicazioni web
- Sono coinvolte differenti figure:
  - System adminstrator: hardening del sistema operativo e dell'infrastruttura IT
  - Code reviewer: hardening del codice applicativo
- Sono coinvolte differenti risorse, non ultimo il core business aziendale

Cosa devo difendere: Application security

- Il problema della sicurezza applicativa viene sottovalutato:
  - Rifiuto del problema: "la mia applicazione è sicura"
  - Gli errori a "runtime" non sono importanti
- La sicurezza applicativa viene ricondotta all'attività di penetration test
  - Rimediare dopo un penetration test richiede un effort non paragonabile a quello necessario se si fosse intervenuto subito
  - Il pericolo è quello di non risolvere il problema in favore di soluzioni tampone

Agenda

- Introduzione
- Cosa devo difendere
  - Application security
- Come lo difendo
  - Design applicativo
  - Strategie di safe coding
  - Code review
- Con cosa lo difendo
  - Orizon
- Conclusioni

Come lo difendo: design applicativo

- Le prime fasi del ciclo di sviluppo sono critiche
- Fase di design
  - Input: il dominio applicativo dove si colloca l'applicazione
  - Output: un modello che individui le parti costitutive del dominio applicativo e le loro connessioni
- Supportare il team di design significa:
  - Verificare l'applicazione dell'imperativo "divide et impera"
  - Verificare le connessioni tra le varie componenti

Come lo difendo: design applicativo

- Le parti del modello vengono tradotte nelle componenti reali
- Fase architetturale:
  - Input: modello del sistema
  - Output: documento infrastrutturale di architettura
- Supportare l'architetto significa:
  - Verificare che le componenti del sistema non siano ridondate

Come lo difendo: safe coding

- (as usual) un codice al 100% sicuro non esiste
- Esistono però numerosi modi per renderlo sicuro
  - Guideline da seguire durante lo sviluppo
  - "Code defensively"
  - Fasi di test rigorose
  - Code review
  - Vulnerability assestment

Come lo difendo: safe coding

- Q: "quanto deve essere sicuro il mio codice?"
- A: "quanto basta"
  - Non devo introdurre meccanismi di protezione che si traducano in una perdita di sicurezza (un sistema troppo complesso di password difficile da seguire per gli utenti)
  - Non devo investire risorse per difendere porzioni di codice non significative o che non trattano dati sensibili

Come lo difendo: safe coding (Java version)

- Evitare le inner class
  - Il compilatore traduce le *inner class* in classi normali accessibili all'interno di un package
  - Una *inner class* ha visibilità su attributi e metodi dichiarati `private` dalla *outer class*
  - Per realizzare questo il compilatore cambia lo scope di questi metodi da `private` a `package`
- Oppure
  - Se proprio devo utilizzare delle *inner class* fare in modo che siano `private`

Come lo difendo: safe coding (Java version)

```
package org.owasp.safecoding;
public class OuterClass {
        private class InnerClass {
        . . .
    }
}
```

DON'T

```
package org.owasp.safecoding;
public class OuterClass {
}
class InnerClass {
        . . .
}
```

DO

Come lo difendo: safe coding (Java version)

- Non utilizzare il nome della classe per confrontare due oggetti
  - Più oggetti all'interno della JVM possono avere lo stesso nome della classe
  - Oggetti possono essere creati ad hoc con un nome di classe legale per soddisfare i miei controlli di uguaglianza
- Oppure
  - Confronto gli oggetti direttamente senza basarmi sul nome della classe

Come lo difendo: safe coding (Java version)

```
public boolean isSameClass(Object o) {
    Class tC = this.getClass();
    Class oC = o.getClass();
    return (tC.getName() == oC.getName());
}
```

DON'T

```
public boolean isSameClass(Object o) {
    Class tC = this.getClass();
    Class oC = o.getClass();
    return (tC == oC);
}
```

DO

Come lo difendo: safe coding (Java version)

- Rendo le mie classi non clonabili
  - Impedisco che un attaccante crei nuove istanze di una classe C1 che ho definito
  - Impedisco che un attaccante crei una sottoclasse C2 di C1 implementando per questa java.lang.Clonable
- Ridefinisco il metodo `clone()`

```
public final Object clone() throws java.lang.CloneNotSupportedException
{
    throw new java.lang.CloneNotSupportedException();
}
```


Come lo difendo: safe coding (Java version)

- Rendo le mie classi non serializzabili
  - Impedisco l'accesso agli stati interni della mia classe impedendo che vi si acceda attraverso una sequenza *raw* di byte
- Ridefinisco il metodo `writeObject()`
  - Lo dichiaro final per impedire un *override*

```
private final void writeObject(ObjectInputOutputStream in) throws
java.io.IOException {
    throw new java.io.IOException("My class can't be serialized");
}
```

Come lo difendo: safe coding (Java version)

- Non dipendere dallo scope del package
- Voglio impedire che un attaccante aggiunga una classe al mio package ed acquisisca visibilità su quanto dichiarato private
- Utilizzo il *package sealing* racchiudendo il mio package all'interno di un *sealed JAR* file.

Come lo difendo: safe coding (Java version)

- Non restituire puntatori ad oggetti non dichiarati come final o in alternativa clonare tali oggetti prima di restituirne il puntatore

```
private Date fDate;
. . .
public Date getDate() {
    return fDate;
}
```

DON'T

```
public Date getDate() {
    return new Date(fDate.getTime());
}
```

DO

Come lo difendo: safe coding (Java version)

- Definire i metodi, gli attributi e le classi come private
  - Esplicito chiaramente gli oggetti che voglio essere visibili all'esterno
- Definire i metodi, gli attributi e le classi come final
  - Impedisco che quanto dichiarato venga esteso da un attaccante

Code Review

- Cambiare approccio durante lo sviluppo per aumentare la qualità del codice applicativo sviluppato
  - Defensive programming
  - Extreme programming
  - Agile programming
- Codice di elevata qualità implica
  - moduli di dimensioni contenute
  - frequenti cicli di test
- Questi approcci non bastano comunque a garantire che il codice sia conforme a standard di sicurezza elevati

Code Review

- Revisionare il codice applicativo implica due approcci distinti:
  - Revisione statica
  - Revisione dinamica
- Entrambi i due approcci sono composti da:
  - Una fase di test automatizzato per evidenziare problemi evidenti oppure imperfezioni nel design dell'applicazione e nella connessione tra moduli differenti
  - Una fase di revisione manuale dove utilizzando i risultati della fase precedente per effettuare controlli più affinati sulla semantica e sull'ingegnerizzazione del codice applicativo

Revisione statica

- Il punto di partenza di una code review è quello di tracciare delle metriche sul codice in esame
- Queste metriche vengono introdotte da una scienza chiamata "Ingegneria del Software"
- La misura della complessità di un modulo software può dare delle informazioni di massima sulla probabilità di errori semantici o di disegno architetturale

Revisione statica

- Alcune metriche utili in questa fase sono:
  - Numero di linee di codice
  - Numero di linee di commento
  - Numero di metodi (per i linguaggio OO)
  - Indice di Complessità ciclomatica (calcola il numero di branch decisionali e istruzioni di loop all'interno di un flusso di controllo)
  - Indice di manutenibilità secondo lo standard del Software Engineering Institute (SEI) della Carnegie Mellon University
  - Numero di classi
  - Numero di metodi

Revisione statica - tools

- Effettura una buona fase di Static code review getta le basi per focalizzare l'intervento manuale del revisore
- E' possibile automatizzare questa fase di calcolo delle metriche del software attraverso vari strumenti tra i quali:
  - JavaMetrics (http://www.semdesigns.com/Products/Metrics/JavaMetrics.html)
  - SmartBear - CodeReviewer (http://www.codehistorian.com/codereviewer-detail.php)
  - M Square Technologies - RMS (http://msquaredtechnologies.com)
  - Jupiter Ecplise Plugin (http://csdl.ics.hawaii.edu/Tools/Jupiter/)
  - SSW Code Auditor (http://www.ssw.com.au/SSW/codeauditor/)
  - Fortify (http://www.fortifysoftware.com)

Revisione statica – Manual review

- Utilizzando i risultati dei test automatizzati il focus della revisione verrà diretto verso:
  - Le classi che accettano input dall'esterno (form html, connessioni ftp, ...)
  - Le classi che producono output (per evitare race condition o DoS verso la memoria di massa disponibile)
  - Le classi con elevato indice di un numero di cicli e costrutti di controllo. Queste classi sono a prima vista più complesse quindi la probabilità di bug che possono portare ad una security issue sono maggiori
  - Le classi con minor numero di commenti. Queste classi probabilmente sono poco manutenute, occorre quindi che siano testate in maniera più intensiva

Revisione statica

- Al termine della fase di revisione statica
  - Verranno evidenziate le carenze di disegno e di infrastruttura tra i moduli che compongono l'applicazione esaminata
  - Verranno evidenziate le porzioni di codice che sono
    - Difficilmente manutenibili
    - Troppo complesse
    - Ridondanti
  - Verranno proposte delle modifiche da apportare al codice e all'infrastruttura per sopperire alle carenze individuate
- Le modifiche verranno discusse col team di sviluppo in un'ottica di confronto costruttivo tra team e revisore

Revisione dinamica

- Temporalmente si colloca prima e dopo la fase di revisione statica
- Va eseguita sui singoli moduli di un'applicazione seguendo un approccio "divide et impera"
- Consiste nell'individuare delle precondizioni e delle postcondizioni che un modulo software deve rispettare
- Scopo della revisione dinamica del codice è
  - Eseguire il modulo software e verificare che quando le precondizioni sono verificate anche le postcondizioni lo sono
  - Eseguire il modulo software e veridicare che quando le precondizioni non sono verificate la situazione di errore viene gestita in maniera corretta

Revisione dinamica

- Occorre individuare in maniera efficace le precondizioni al modulo software da testare; in questo modo so calcolare gli input che non sono attesi
- Il modulo software deve reagire ad input che violano le precondizioni con comportamenti deterministici e che non minino la sicurezza e la stabilità del modulo
- La revisione dinamica rappresenta una sorta di test funzionale e di collaudo di un modulo software.
- Automatizzare questa fase è più complesso
- Richiede la scrittura di un applicativo di test ad hoc che dipende
  - Dalla tecnologia utilizzata
  - Dal tipo di modulo che si sta testando

Code Review

- Combinando un approccio di revisione statica ad un approccio di revisione dinamica
  - Aumentiamo il livello di qualità del codice
  - Miglioriamo l'ingegnerizzazione dello stesso
  - Miglioriamo la riusabilità e semplifichiamo il codice dei singoli moduli software
  - Verifichiamo tra due sessione di scrittura del codice che il comportamento del modulo rimane coerente

Q&A

Riferimenti