DuD
Datenschutz und Datensicherheit

Vulnerability of Code

Security Design Flaws

Petra Barzin

Der Wettlauf gegen die Zeit, um Sicherheitslöcher zu finden, bevor das einem Angreifer gelingt, ist kein befriedigender Ansatz, um Vertrauen in die Sicherheit von Software zu wecken. Nicht diese Symptome, sondern die Ursachen müssen bekämpft werden, damit Sicherheitslöcher möglichst erst gar nicht entstehen. Der Beitrag zeigt, wie sich in der Beseitigung kostspielige Design-Fehler vermeiden lassen.

Dipl.-Inform. (FH) Petra Barzin

Secorvo Security Consulting GmbH; Sichere Softwareentwicklung, Single-Sign-On-Lösungen, E-Mail-Sicherheitslösungen, Public-Key-Infrastrukturen, Digitale Signaturen

E-Mail: petra.barzin@secorvo.de

1 Einleitung

Veröffentlichungen von Security Patches für gefährliche Sicherheitslücken gehören heutzutage schon fast zur Tagesordnung und rufen kaum mehr Schrecken oder gar Protest in der Öffentlichkeit hervor. Dabei sind diese Sicherheitslöcher nicht auf einige wenige Produkte beschränkt, sondern alle gängigen Standardanwendungen sind von ihnen betroffenen. Zur Behebung der Schwachstellen stellen die Hersteller zwar kostenlos Security Patches zur Verfügung, aber das Einspielen von Patches verursacht zusätzliche Kosten und birgt außerdem Risiken für Stabilität und Einsatzfähigkeit der betroffenen Anwendung. Eventuell enthält ein Security Patch sogar neue Sicherheitslöcher.

Sicherheitslöcher werden von Angreifern verwendet, um unternehmenskritische Daten zu stehlen, Viren und Würmer zu verbreiten oder Computersysteme zu sabotieren. Firewalls oder Intrusion Detection Systeme sind schon lange nicht mehr als Schutz ausreichend, da sie Angriffe auf die Anwendung selber nicht verhindern können. So kann z. B. eine Firewall nicht entscheiden, ob ein Eingabeparameter bei einer Anwendung gültig ist oder einen „code injection“ Angriff darstellt. Dieses Wissen hat nur die jeweilige Anwendung selber.

Folglich müssen Angriffe aufgrund von Sicherheitslücken schon bei der Entwicklung einer Anwendung möglichst ausgeschlossen werden. Leider wird in den meisten Fällen dem Thema „Sicherheit“ im Software Entwicklungsprozess (Software Development Lifecycle, SDLC) weder an den Universitäten noch später im Berufsalltag die nötige Beachtung gewidmet, um frühzeitig der Entstehung von Sicherheitslücken entgegenzuwirken.

Ursachen für Sicherheitslücken gibt es viele. Zunächst sind sich die am Software-Entwicklungsprozess beteiligten Personen in der Regel nicht über die Gefährdungen bewusst. Es fehlen die nötige Security Awareness und geeignete Schulungen, um Projekt- und Entwicklungsleiter, Softwarearchitekten, Entwickler und Tester für das Thema zu sensibilisieren und ihren Blick für Sicherheit zu schärfen.

Bei der Software-Entwicklung selber führen Fehler im Design und der Kodierung zu einer Vielzahl verschiedener Angriffsmöglichkeiten. Und nicht zuletzt beim Betrieb sind Fehler in der Konfiguration von IT-Systemen die Ursache für Schwachstellen und darauf aufbauenden Exploits.

2 Angriffe

Angriffe lassen sich nach Software-Entwicklungsphasen unterscheiden, aus denen die entsprechenden Schwachstellen resultieren. Angriffe basierend auf Architektur- und Designschwächen sind dabei die am aufwändigsten zu behebenden Schwachstellen, da im Software-Entwicklungsprozess bis zur Designphase zurückgegangen werden muss, um durch ein geändertes Design die Sicherheitslücke zu beheben. Angriffe auf Implementierungsebene hingegen lassen sich leichter beheben als Architektur- und Designfehler. Angriffsmöglichkeiten im Betrieb können häufig durch einfache Konfigurationsänderungen beseitigt werden. Tabelle 1 gibt einen Überblick über die verschiedenen Angriffstypen, aufgeteilt nach den verschiedenen Entwicklungsphasen, aus denen sie resultieren.

2.1 Architektur und Design

Beim Man-in-the-middle-Angriff werden Verbindungsanfragen auf den Angreifer umgeleitet, ohne dass Sender und Empfänger dies bemerken. Der Angreifer spiegelt sowohl dem Sender als auch dem Empfänger vor, der jeweils andere Kommunikati-

Angriffe auf Architektur- und Designebene	Angriffe auf Implementierungsebene	Angriffe im Betrieb
Man-in-the-middle	Buffer overflow	Denial-of-service
Replay attack	Back door attack	Defaults accounts attack
Sniffer attack	Code injections	Password cracking
Session hijacking		

Tabelle 1: Angriffstypen

onspartner zu sein. So kann er sowohl die übertragenen Daten mitlesen als auch diese unbemerkt verändern. Möglich wird dieser Angriff dadurch, dass die Identität des Kommunikationspartners von der Anwendung nicht überprüft wird, d. h. dass keine Authentisierung der Kommunikationspartner stattfindet – ein klarer Fehler im Design der Anwendung.

Bei einer Replay-Attacke hört der Angreifer bestimmte Daten mit, wie z. B. ein übertragenes Passwort, und verwendet dieses anschließend selber, um sich bei dem Server anzumelden. Hierbei schützt auch eine Verschlüsselung des Passwortes nicht vor diesem Angriff, da der Angreifer selber das Passwort gar nicht kennen muss. Es reicht, das verschlüsselte Passwort erneut an den Server zu senden. Möglich wird dieser Angriff dadurch, dass das verschlüsselte Passwort immer gleich aussieht. Durch die Einführung von Nonces, die nach einmaliger Verwendung verworfen werden, können Replay-Attacken beispielsweise verhindert werden. Die Verwendung solcher Nonces muss schon beim Design der Anwendung berücksichtigt werden.

Eine Sniffer-Attacke ist eine passive Angriffstechnik ohne Veränderung von Daten. So kann z. B. ein spezieller Passwortsniffer, der die typischen Netzwerkprotokolle wie FTP, HTTP, POP3 oder andere kennt, zum Filtern und Ausspähen von Benutzername und Passwort verwendet werden. Möglich ist dieser Angriff, wenn kein verschlüsselter Kommunikationskanal verwendet wird. So muss schon beim Design der Anwendung die Verschlüsselung der Kommunikation vorgesehen werden.

Das Session hijacking bezeichnet einen Angriff, bei dem eine bestehende Verbindung mit dem Ziel übernommen wird, durch die „Entführung“ dieser Sitzung die Vertrauensstellung auszunutzen, um dieselben Privilegien wie der rechtmäßig authentifizierte Benutzer zu erlangen. Ein typisches Beispiel hierfür sind Webanwendungen mit ihrer Sitzungsverwaltung über Session-IDs. Die Sammlung notwendiger Informationen, wie z. B. die Session-ID, erfordert eine vorangehende Sniffing- oder Man-in-the-middle-Attacke. Session-IDs werden meistens über ein GET- oder POST-Argument oder über ein Cookie an den Server übermittelt und sind somit angreifbar. Am leichtesten lässt sich daher Session hijacking verhindern, indem bereits das Ausspionieren der notwendigen Informationen durch verschlüsselte Kommunikationsverbindungen verhindert wird. Dies muss schon bei Architektur und Design der Anwendung berücksichtigt werden.

2.2 Implementierung

Ziel von Buffer-overflow-Angriffen ist es, Angriffscode auf Zielsystemen auszuführen. Das wohl bekannteste Beispiel für einen Buffer-overflow-Angriff ist der Code-Red-Wurm aus dem Jahr 2001. Möglich wird dieser Angriff durch Fehler in der Programmierung, wenn zu große Datenmengen in einen dafür zu kleinen Speicherbereich geschrieben werden können. Dies kann von Angreifern ausgenutzt werden, um die Rücksprungadresse eines Unterprogramms mit beliebigen Daten zu überschreiben, und so seinen übermittelten Angriffs-Maschinencode mit den Privilegien des für den Buffer overflow anfälligen Prozesses ausführen zu können.

Besonders gefährdet sind mit der Programmiersprache C und C++ entwickelte Programme, aber auch in anderen Programmiersprachen können Buffer overflows nicht ausgeschlossen werden. Zur Vermeidung solcher Angriffe müssen bei der Programmierung immer genau die Feldgrenzen eines „Arrays“ überprüft werden.

Bei der Back-door-Attacke macht sich der Angreifer eine Hintertür zunutze, um in die Anwendung einbrechen zu können. Eine Anwendung ist nur so sicher wie ihr schwächstes Glied: Es reicht nicht, das Haupteingangstor mit einer Zugbrücke und Soldaten zu bewachen, wenn eine Hintertür zu einem unbewachten und ungesicherten Geheimgang existiert. Ein Entwickler darf auf gar keinen Fall solche Hintertüren in seinem Programmcode einbauen – auch wenn sie noch so praktisch zum Entwickeln und Testen sein mögen. Selbst wenn solche Hintertüren nirgends dokumentiert werden, findet sie ein Angreifer früher oder später doch.

Die wohl bekanntesten Code-injection-Angriffe sind die SQL injections, bei denen ein Angreifer versucht, über die Anwendung, die den Zugriff auf die Datenbank bereitstellt, durch den gezielten Einsatz von Funktionszeichen weitere SQL-Befehle einzuschleusen. Sein Ziel ist es dabei, Daten in seinem Sinne zu verändern oder Kontrolle über den Server zu erhalten. So kann folgende SQL-Abfrage ohne Überprüfung der Eingabe durch die Anwendung leicht von einem Angreifer missbraucht werden:

```
select count(*) from users
where UserName='admin' AND
UserPassword='test'
```

Meldet sich nun nicht der Administrator mit seinem Passwort „test“ an, sondern der Angreifer, der das Passwort des Administrators nicht kennt, so gibt dieser in der Eingabemaske als Benutzername „admin“ und das folgende Passwort ein: ' OR '1' = '1. Dann sieht der SQL String wie folgt aus:

```
select count(*) from users
where UseName='admin' AND
UserPassword='' OR '1'='1'
```

Obwohl dem Angreifer das Administrator-Passwort nicht bekannt ist, erhält er durch die Ergänzung des booleschen Ausdrucks OR'1'='1' immer eine wahre Aussage und somit Zugang zu dem System. Verhindert werden können solche Code-injection-Angriffe durch konsequente Eingabeüberprüfung und korrekte Output-Kodierung.

2.3 Betrieb

Bei einem Denial of Service oder kurz DoS-Angriff versucht ein Angreifer einen Server außer Betrieb zu setzen, indem er ihn mit so vielen Anfragen überlastet, dass das System die Aufgaben nicht mehr bewältigen kann und zusammenbricht. Kommen die Angriffe gleichzeitig von einer Vielzahl unterschiedlicher Systeme, so spricht man von einem Distributed Denial of Service (DDoS)-Angriff. Zur Abwehr eines DoS-Angriffs sollten Verfügbarkeit und Zuverlässigkeit der Systeme erhöht werden.

Ein Defaults-accounts-Angriff macht sich den Umstand zu Nutze, dass viele Anwendungen mit vorbelegten Standard-Nutzern und Passwörtern ausgeliefert werden, die häufig bei der Installation von den zuständigen Administratoren nicht geändert,

sondern in ihrer Default-Konfiguration belassen werden. So hat ein Angreifer ein leichtes Spiel, um sich mit Administratorrechten Zugang zu dem System zu verschaffen. Daher müssen Default-Passwörter bei der Installation und Konfiguration eines neuen Systems immer geändert werden.

Beim Password cracking ist ein Angreifer in der Lage, das Passwort eines Benutzers zu ermitteln. Hierbei gibt es verschiedene Angriffsmöglichkeiten: Brute-Force, codebook attack, dictionary attack, Social Engineering oder Phishing. Grundlage für ein codebook attack ist ein Codebuch mit Zuordnung von Chiffretexten und Klartexten. Ein dictionary attack (Wörterbuchangriff) ist eine Methode der Kryptoanalyse, um ein unbekanntes Passwort mit Hilfe einer Passwortliste zu knacken. Passwort-Cracker finden ein achtstelliges Passwort aus Groß- und Kleinbuchstaben in wenigen Stunden bis Tagen; stammt das Passwort aus dem deutschen Sprachschatz, ist es sogar in Sekunden geknackt.

3 Sicheres Design

Wie in Abschnitt 2 dargestellt sind Angriffe, die auf Architektur- und Designschwächen basieren, die am aufwändigsten zu behebenden Schwachstellen. Daher geht dieser Beitrag speziell auf diese Software-Entwicklungsphase ein, um aufzuzeigen, mit welchen Maßnahmen die Sicherheit einer Anwendung in der Architektur- und Designphase gesteigert werden kann. Die anderen Phasen des Software Development Lifecycles (SDLC) werden nicht näher betrachtet, sondern es wird auf die weiterführende Literatur verwiesen, wo Maßnahmen zur sicheren Anwendungsentwicklung bezogen auf den gesamten SDLC beschrieben sind.

Alle Maßnahmen zur sicheren Anwendungsentwicklung sind unabhängig von der Entwicklungsmethode; d. h. unabhängig davon, ob beispielsweise das Wasserfallmodell, Prototyping Model, Rational Unified Process (RUP) oder Extreme Programming (XP) eingesetzt wird, können diese Maßnahmen in das Vorgehensmodell zur Programmentwicklung eingebunden werden.

3.1 Bedrohungsanalyse

Zu Beginn der Architektur- und Designphase sollte eine Risiko- und Bedrohungsanalyse durchgeführt werden. Das Finden von Bedrohungen erfordert eine besondere Kreativität, um sich in die Lage eines Angreifers zu versetzen und sämtliche möglichen Bedrohungen zu identifizieren. Besonderes Augenmerk sollte hierbei auf die Mechanismen zur Benutzerauthentifizierung, das Session Management für Webanwendungen, die präventiven Maßnahmen zum Schutz vor Parameter-Manipulationen, die Offenlegung sensibler, unternehmenskritischer Daten, die Berechtigungsstrategie und die Erhaltung der Datenintegrität gerichtet sein.

Als eine Hilfestellung zur Ermittlung von Bedrohungen dient der STRIDE Ansatz von Microsoft – der auch von dem Open Web Application Security Project (OWASP) empfohlen wird, um Sicherheitslücken im Design aufzudecken. STRIDE ist eine Abkürzung, die sich aus den folgenden Bedrohungen ergibt:

- ***S**poofing identity*: Vortäuschen einer falschen Client oder Server Identität
- ***T**ampering with data*: Verfälschung von Daten
- ***R**epudiation*: Möglichkeit zur Abstreitung von Benutzeraktionen
- ***I**nformation disclosure*: Offenlegung von Informationen für nicht berechtigte Nutzer
- ***D**enial of Service*: Verweigerung von Diensten für berechtigte Nutzer
- ***E**levation of privilege*: Aufdeckung von Informationen für nicht berechtigte Nutzer durch Anhebung seiner Privilegien

3.2 Sicherheitsarchitektur

Die Sicherheitsarchitektur beschreibt das zugrunde liegende Sicherheitsmodell und eine übergreifende Perspektive zum Sicherheitsdesign der Software. Wenn vorhanden, beschreibt es auch den Funktionsumfang und die Programmierschnittstellen eines zentralen Security Frameworks im Unternehmen oder in der Organisation. Außerdem wird in der Sicherheitsarchitektur erläutert, wie die zu entwickelnde Anwendung den zuvor identifizierten Bedrohungen entgegen wirkt.

3.3 Richtlinien

Mit Hilfe einer überschaubaren Anzahl von Richtlinien für sichere Softwareentwicklung lassen sich grundsätzliche Sicherheitsprinzipien einer Anwendung in einem Unternehmen bzw. einer Organisation verankern. Diese Sicherheitsprinzipien können sowohl das Design, die Implementierung als auch das Testen betreffen. Die bekanntesten Beispiele für solche Richtlinien sind die CLASP Security Principles (Comprehensive, Lightweight Application Security Process) von OWASP und im deutschsprachigen Raum „Die 10 Goldenen Regeln der IT-Sicherheit“, die in dem Forschungsprojekt secologic von der Commerzbank und SAP entwickelt wurden.[1] Typische Sicherheitsprinzipien sind beispielsweise:

- *Least Privilege*: Das Prinzip der minimalen Rechtevergabe fordert nur die allernötigsten Rechte zu vergeben
- *Defense-in-Depth*: Redundante Sicherheitsmechanismen erhöhen die Sicherheit
- *Secure-by-Default*: Default Einstellungen sollen so sicher wie möglich sein
- *Fail-safe*: Bei einem Fehler in der Anwendung darf kein unsicherer Zustand auftreten
- *Input Validation*: Alle Eingabedaten müssen sorgfältig auf Gültigkeit überprüft werden, bevor sie weiterverarbeitet werden.

3.4 Sicherheitsbetrachtungen

Im Grob- und Feinkonzept sollen nicht nur funktionale Beschreibungen enthalten sein, sondern auch eine Sicherheitsbetrachtung, ob und inwiefern eine Komponente sicherheitsrelevant ist, d. h. die Sicherheit der Software beeinflusst. Eine solche Beschreibung kann in Anlehnung an den RFC 3552 „Guidelines for Writing RFC Text on Security Considerations“ erstellt werden.

Im Feinkonzept sollte der zu implementierende Sicherheitsmechanismus detailliert beschrieben werden, um dem Entwickler klare Vorgaben zu machen. Wenn ein zentrales Security Framework im Unternehmen oder in der Organisation vorhanden ist, sollten immer die bereits dort implementierten Sicherheitsfunktionen verwendet werden, anstatt einen Sicherheitsmechanismus selber neu zu entwickeln.

3.5 Software-Angriffsfläche

Eine frühzeitige Definition der minimalen Software-Angriffsfläche und eine fortlaufende Überprüfung während der Entwicklung hilft, die Anzahl der Sicherheitslücken in der Anwendung zu verringern. Treten

[1] Siehe Giesecke/Fünfrocken, in diesem Heft.

z. B. während der Entwicklung zusätzliche offene Ports auf, die nicht im Design spezifiziert sind, sollte kritisch hinterfragt werden, warum diese Vergrößerung der Software-Angriffsfläche nötig ist. Um die Software-Angriffsfläche möglichst klein zu halten, sollte auch die Menge an „Running Code" kritisch überprüft und ggf. gemäß der 80/20 Regel reduziert werden: Benötigen 80 Prozent der Anwender diese Funktionalität? Wenn nicht, sollte die Funktion bzw. der Dienst deaktiviert werden.

Weiterhin sollte der Netzwerkzugang so eingeschränkt werden, dass entweder der Netzwerkzugang durch die Anwendung auf das lokale Subnetz oder einen definierten IP-Bereich beschränkt ist, oder der Zugriff auf das Netzwerk nur nach erfolgreicher Benutzerauthentifizierung erfolgt.

Eine weitere Sicherheitsmassnahme zur Verringerung der Software-Angriffsfläche ist die Kontrolle und ggf. Reduktion der Privilegien, unter denen ein Code ausgeführt werden muss. Hier gilt das Sicherheitsprinzip „least privilege".

Fazit

Massnahmen zur Steigerung der Sicherheit müssen im gesamten Software Development Lifecycle (SDLC) ergriffen werden, angefangen bei der Anforderungsanalyse bis hin zur Auslieferung der fertigen Software.

Je früher im SDLC Schwachstellen in einer Anwendung erkannt werden, desto leichter und schneller können sie behoben werden. Deshalb ist es besonders wichtig, eine Anwendung frühzeitig und genau auf mögliche Architektur- und Designschwächen zu überprüfen. Sichere Softwareentwicklung erfordert sicherheitsbewusste und gut ausgebildete Software Architekten und Programmierer.

Literatur

[DOW] Mark Dowd, John McDonald, Justin Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*, November 10, 2006, Addison-Wesley Longman Verlag

[HER] Shawn Hernan, Scott Lambert, Tomasz Ostwald, Adam Shostack, *Threat Modeling – Uncover Security Design Flaws Using The STRIDE Approach*, MSDN magazine, November 2006, http://msdn.microsoft.com/msdnmag/issues/06/11/ThreatModeling/

[HOWa] Howard, Michael, *Attack Surface: Mitigate Security Risks by Minimizing the Code You Expose to Untrusted Users*, MSDN Magazine, November 2004, http://msdn.microsoft.com/msdnmag/issues/04/11/AttackSurface/default.aspx

[HOWb] Michael Howard, *A Look Inside the Security Development Lifecycle at Microsoft*, MSDN Magazine, November 2005, http://msdn.microsoft.com/msdnmag/issues/05/11/SDL/default.aspx

[JUS] Elfriede Dustin, *The Secure Software Development Lifecycle*, November 2006, http://www.devsource.com/print_article2/0,1217,a=193825,00.asp

[LIP] Steve Lipner, Michael Howard, *The Trustworthy Computing Security Development Lifecycle*, März 2005, http://msdn2.microsoft.com/en-us/library/ms995349.aspx

[MEI] J.D. Meier, Alex Mackman, Blaine Wastell, Prashant Bansode, Kishore Gopalan, *Security Engineering Index*, August 2005, MSDN, http://msdn2.microsoft.com/en-us/library/ms998404.aspx

[SAP] SAP: *Sicheres Programmieren – Einführung in die sichere Anwendungsentwicklung*, in Zusammenarbeit mit Microsoft, 2005, https://www.sicher-im-netz.de/content/sicherheit/ihre/software/sicheresoftware/download/SichereProgrammierung.pdf

[SEC] secologic: *Die 10 Goldenen Regeln der IT-Sicherheit*, Dezember 2006, http://www.secologic.de/downloads/software/070205_10GoldenRules_SAP_CoBa_V1.pdf

[STO] Gary Stoneburner, Clark Hayden, Alexis Feringa, *Engineering Principles for Information Technology Security (A Baseline for Achieving Security)*, NIST Special Publication 800-27 Rev A, Juni 2004, http://csrc.nist.gov/publications/nistpubs/800-27A/SP800-27-RevA.pdf

[TFS] Task Force on Security across the Software Development Lifecycle, *Improving security across the software development lifecycle*, April 2004, http://www.cyberpartnership.org/SDLCFULL.pdf

[VIE] John Viega, *Security in the software development lifecycle*, Oktober 2004, http://www-128.ibm.com/developerworks/rational/library/content/RationalEdge/oct04/viega/index.html