Sieben Gebote, um Sicherheitslücken beim PHP-Programmieren zu erkennen

Löcher für Schweine

Ursächlich für die Kompromittierung eines Webservers sind entgegen landläufiger Meinung nicht fiese Angreifer, sondern jene, die die Software für das Gerät entwickeln und betreiben. Schlampig getesteter Code bohrt die Löcher, durch die Fremde einsteigen – folgende sieben Gebote helfen. Tobias Eggendorfer

Angriffe auf Websysteme erfolgen heute kaum mehr durch Schwächen in Netzwerkprotokollen, sondern wegen der Fehler in Anwendungen. Viele der spektakulären Hacks der letzten Jahre, etwa der ins Sony-Playstation-Netzwerk, nutzten Programmierfehler in Webanwendungen aus. Die Löcher sind selten exotisch und lassen sich ganz wenigen Kategorien zuordnen, der Sony-Hack zum Beispiel gelang mit einer SQL-Injection. In loser Folge hatte das Linux-Magazin an produktiven Sites Angriffsvektoren und die ursächlichen Programmierfehler herausgearbeitet ([1], [2]).

Zwar stellen moderne Betriebssysteme aufwändige Schutzmaßnahmen gegen Schwachstellen bereit, etwa die Address Space Layout Randomization. Aber auch die hebeln versierte Angreifer mit ein paar Tricks aus. Bleibt allein übrig, die Webapplikationen selbst ohne Sicherheitslücken zu entwickeln. Programmfehler systematisch zu vermeiden ist darum hehres Ziel jedes ernst zu nehmenden Software-Qualitätsmanagements.

Das beginnt lange vor der ersten Programmzeile: Schon in der Konzeptionsphase muss das Projekt überlegen, welche Sicherheitsprobleme auftreten können, welche Sicherheitsanforderung es einhalten will und unter welchen Bedingungen die Software zum Einsatz kommen wird. Spätestens hier sollte das Projekt Codingstandards vereinbaren, damit bei Tests und Codereviews alle die gleiche Sprache sprechen.

Ingenieure wissen, dass Qualität ein Ergebnis des Herstellungsprozesses ist und nicht nach dem Abschluss hineinprüfbar. Open BSD als Positivbeispiel gilt dank klarer Qualitätsstandards und Tests vielen als sicherstes Betriebssystem. Von Anfang an müssen alle an einem Projekt beteiligten Entwickler auf sicheren und fehlerfreien Code achten. Neben regelmäßiger Fortbildung hat sich eine Sammlung von fertigen Routinen für Standardfälle als günstig erwiesen. Wer die hat, erspart es sich, später das Rad neu zu erfinden.

1. Nichts zweimal erfinden

Neue Räder und neue Software laufen zu Beginn längst nicht so rund wie erprobte. Bei Sessionmanagement und Login-Prüfungen schreiben viele Entwickler ihre eigenen Routinen und öffnen damit Session-Surfing oder SQL-Injections Tür und Tor (**Abbildungen 1 und 2**).

Zwei ganz einfache Maßnahme schützen vor solchen Neuerfindungen: Zum einen helfen Absprachen innerhalb des Projekts, wer welche Routinen aus welchen Bibliotheken übernimmt. Zum anderen hebt das gegenseitige Code-Auditieren zweier Entwickler die Qualität. Absprachen senken die Wahrscheinlichkeit für Doppelerfindungen, weil oft einer der Beteiligten eine Lösung für ein Standardproblem schon kennt.

Gleichzeitig führen solche Absprachen zu einem zweiten qualitätssichernden Effekt: Wer seinen Code einem anderen erläutern muss, wird ihn besser dokumentieren und auch gründlicher durchdenken, um gut dazustehen. Außerdem lassen sich die Ergebnisse der gegenseitigen Code-Vorstellungen direkt für die Dokumentation verwenden – eine Sparte, die oft stiefmütterlich behandelt wird.

2. Code-Review betreiben

Solche Absprachen sind weit entfernt von systematischen Code-Reviews, bei denen ein zweiter Entwickler oder ein ganzes Team sich Zeile für Zeile durch das Programm eines Kollegen tastet. Dieses sehr zeitaufwändige Testverfahren bietet – ein gutes Team vorausgesetzt – eine sehr

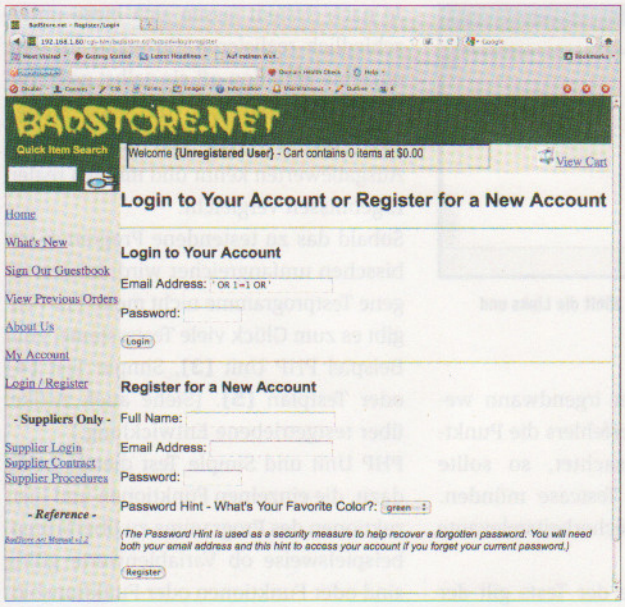

Abbildung 1: Gute Tests hätten ans Licht gebracht, dass die Login-Maske des (Security-Lehr-)Shops Badstore.net SQL-Injections ermöglicht ...

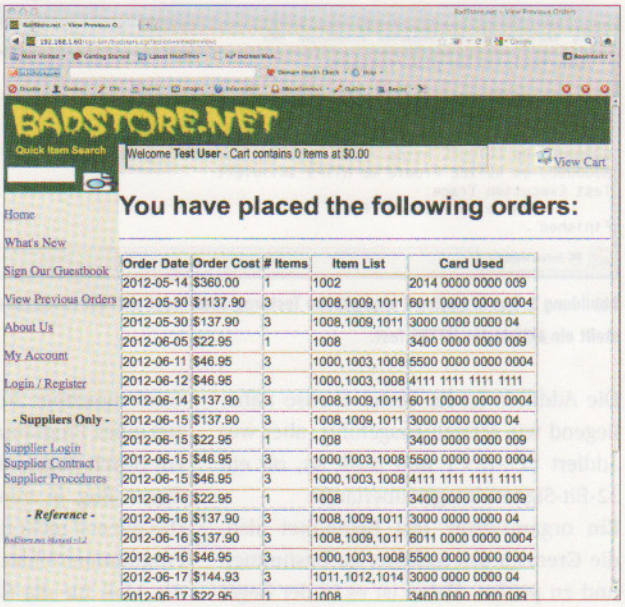

Abbildung 2: ... und so jeden sehen lässt, was Kunden zuvor bestellt haben. Die harmlose Trockenübung funktioniert bei erschreckend vielen echten Shops auch.

hohe Erfolgsaussicht, Fehler zu finden. Open BSD betreibt diesen Aufwand.

3. Datenflussanalyse prüft Eingabedaten

Auf der Suche nach Sicherheitslöchern hilft eine Datenflussanalyse im Rahmen des Code-Review. Dabei gilt es, für jedes eingegebene Datum zu prüfen, ob und wie es Einfluss auf welche anderen Programmaktionen hat. Das ist für einfache Mailformulare simpel, bei Datenbankanwendungen aber sehr zeitintensiv. Aber nur so lassen sich mit etwas Erfahrung Cross Site Scripting, Code-Injections und vergleichbare Angriffsstellen schnell lokalisieren.

Interessant bei solchen Datenflussanalysen sind immer Grenzwerte: Überschreitet etwa eine Integervariable den Wert »PHP_INT_MAX«, führt PHP einen automatischen Typecast nach Float durch. Das kann zu Überraschungen führen, wenn der Wert in ein Datenbankfeld vom Typ Integer wandert. Insbesondere bei Datenbanksystemen wie MySQL, die auch Datentypen wie »tinyint« kennen, geschehen Integer-Overflows schnell.

Bei einer Datenflussanalyse behalten die Entwickler außerdem alle Berechnungen und Manipulationen von Eingabendaten im Programm in Bezug auf Grenzwerte im Auge und erkennen entsprechende Fehler daher frühzeitig.

Mehr noch als Absprachen haben intensive Reviews und Datenflussanalysen den Effekt, dass die Entwickler nicht nur Fehler finden, sondern sich als Team sensibilisieren für kritische Programmierfehler und sich die Dokumentation verbessert. Die Lerneffekte führen mittelfristig zu höherer Codequalität und weniger Fehlern.

4. Testsysteme richtig konfigurieren

Im Zuge einer Datenflussanalyse fallen fehlende URL-Encodings oder doppeltes Escaping von SQL-Statements auf. Multiple Escapes können übrigens auch durch die kleine Unachtsamkeit entstehen, dass in der eigenen Testumgebung PHP mit deaktivierten Magic Quotes läuft und auf dem Produktivsystem mit aktivierten. Deshalb sollte man Produktiv- und Testsystem konsequent mit gleichen Versionen und Konfigurationsdateien betreiben. Ein Cronjob, der ein einfaches Diff über eine SSH-Verbindung laufen lässt, kann dies prüfen.

5. An Debug- und Fehlermeldungen denken

Ein gewisses Sicherheitsproblem bilden vergessene Debugmeldungen. Je detaillierter sie ausfallen, desto mehr nutzen sie dem Angreifer. So erfährt er zum Beispiel Datenbank- und Tabellennamen, auch der in [1] beschriebene Angriff lief über diese Schiene. Gegen vergessene Debugmeldungen hilft ein vereinbartes, vorangestelltes Schlüsselwort, das sich über Grep leicht finden lässt. Alternativ binden Entwickler auf ihrem Testsystem eine spezielle Debugausgabe-Funktion per Include ein, die auf dem Produktivsystem leer bleibt.

Wie Debug- helfen Angreifern auch Fehlermeldungen, die nach außen dringen. Normale Benutzer verwirren technische Meldung außerdem, was die Usability beeinträchtigt. Besser sind detaillierte Logeinträge oder automatische Mails an den Admin. In eine gute Dokumentation, sie ist eine Voraussetzung für Softwaretests, gehört eine Liste möglicher Fehlermeldungen und unter welchen Bedingungen die Webapplikation sie erreicht. Damit kann ein Tester gezielt Fehler provozieren und so das Verhältnis von Nutzerfreundlichkeit und Sicherheitsrisiko einschätzen.

6. Testfälle konstruieren

Anhand der bisherig gewonnen Erkenntnisse lassen sich Testfälle beschreiben. Das sind Kombinationen aus definierten Ausgangssituationen und einem bestimmten vorhersagbaren Verhalten, das ein Programm daraufhin zeigen soll. So wäre ein Testcase für einen Taschenrechner »2 + 2« mit der erwarteten Ausgabe »4«.

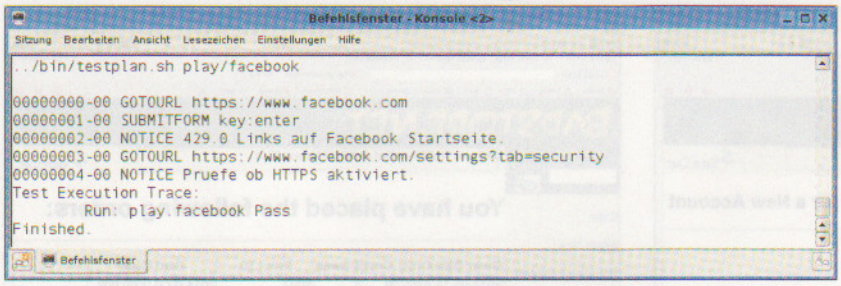

Abbildung 3: Das in Listing 2 angegebene Testprogramm loggt sich auf Facebook ein, zählt die Links und stellt ein aktiviertes HTTPS fest.

Die Addition ist für Tester ebenso naheliegend wie schnell ausgeführt, aber wer addiert $(2^{31}-1)+1$ und prüft so, ob ein 32-Bit-Signed-Integer überläuft?

Ein organisierter Test beinhaltet also, die Grenzen des Systems zu bestimmen und zu prüfen. Dabei ist es in der Regel nicht zielführend, wenn jene Entwickler die Testfälle definieren, die auch die Anwendung geschrieben haben – ihre große Nähe zum Code macht es wahrscheinlich, dass sie betriebsblind sind und ihre eigenen Fehler übersehen.

Tests sind umfangreich, für jede Funktion selbst, für jedes Modul und für das fertige System gibt es Testfälle. Sie sollten bewusst auch Grenzsituationen abdecken. Ebenso sollten Entwickler Tests für jeden bekannten Bug integrieren. Hat also der Taschenrechner früher irgendwann wegen eines Programmierfehlers die Punkt-vor-Strich-Regel missachtet, so sollte dieser Bug in einen Testcase münden. Das gilt erst recht für sicherheitsrelevante Programmierfehler.

Als Maß für die Güte der Tests gilt der Anteil des berücksichtigen Codes. Eine 100-Prozent-Überdeckung ist wünschenswert, aber nicht immer möglich. In ereignisgesteuerten Programmen oder bei Exception-Handlern für schwer provozierbare Fehler, bleibt mancher Zweig praktisch unerreichbar.

7. Testprogramme benutzen

Das alles zeigt: Neben manuellen werden automatisierte Tests unumgänglich. Je nach System lassen sich diese unterschiedlich realisieren. So reicht für eine einzelne Funktion oft ein selbst geschriebenes Testprogramm, das eine Menge von Eingabewerten und zugeordneten Ausgabewerten kennt und mit den realen Ergebnissen vergleicht.

Sobald das zu testendene Programm ein bisschen umfangreicher wird, lohnen eigene Testprogramme nicht mehr. Für PHP gibt es zum Glück viele Testsysteme, zum Beispiel PHP Unit [3], Simple Test [4] oder Testplan [5]. (Siehe auch Artikel über testgetriebene Entwicklung.)

PHP Unit und Simple Test dienen beide dazu, die einzelnen Funktionen und Instruktionen des Programms zu überprüfen, beispielsweise ob Variablenwerte gültig sind oder Funktionen oder Funktionsblöcke korrekte Werte zurückliefern. Daher sind beide Umgebungen nur für PHP geeignet. Das Handbuch [6] zu PHP Unit bietet eine sehr gute Einführung in das automatisierte Testen – Wissen, das problemlos auf Simple Test übertragbar ist, dessen Dokumentation deutlich magerer ist. Funktional und konzeptionell sind beide vergleichbar.

Testplan dagegen prüft, ob das Programm auf vorgegebene Nutzereingaben erwartete Werte zurückgibt; es testet also die

Testplan in der Praxis

Die Installation von Testplan ist relativ simpel und auf der Homepage [5] gut erklärt. Nur fehlt der Hinweis, die Umgebungsvariable »TESTPLAN_HOME« korrekt zu setzen, im Beispiel zu diesem Artikel mit:

```
export TESTPLAN_HOME=~/testplan-1-0-r6/
```

Auf dem Fedora-Testrechner des Autors war es nicht notwendig, »JAVA_HOME« zu setzen.

Der erste, einfache Test in Listing 1 prüft die Zahl der Links auf der Homepage des Autors und demonstriert dabei gleich mehrere Elemente von Testplans Skriptsprache: Sie beherrscht Schleifen, kann zählen, Dinge aus der Ausgabe extrahieren und selbst Ausgaben erzeugen. So selektiert Zeile 5 die Links aus der Antwort (»%Response%«) über den Parameter »a«, das entspricht dem HTML-Tag für einen Link. Der »href«-Parameter in Zeile 6 zeigt auf die URL, die Zeile 7 ausgibt.

Das Listing 2 enthält den einfachen Zähltest aus Listing 1, arbeitet aber komplexer: Hier meldet sich Testplan für den Autor bei Facebook an und überprüft, ob HTTPS für Facebook aktiviert ist. Das geschieht zweistufig: Zunächst stellt die Funktion »Check« sicher, dass auch der gewünschte Text auf der Seite steht. Im Erfolgsfall läuft das Testskript weiter. Es liest dann die Meldung aus, entfernt das HTML daraus und prüft mit einer Regular Expression, ob dort »aktiviert« steht.

Wenn ja, meldet das Skript diesen Test wie in Abbildung 3 als bestanden (»Pass«). Andernfalls versucht der Test, HTTPS selbsttätig nachzuaktivieren (Abbildung 4). Dazu ist ein kleiner Kunstgriff nötig, denn Facebook verpasst den Formularen bei jedem Zugriff eine neue, scheinbar zufällige ID. Dem Xpath-Ausdruck in Zeile 30 gelingt es, die ID zu lesen. Das Ergebnis wandert in die lokale Variable »%id%«. Ebenso wie der Inhalt eines verdeckten Eingabefelds in Zeile 31. Mit diesen gewonnenen Werten lässt sich das gewünschte Formular auf der Seite adressieren (Zeile 33) und korrekt ausgefüllt an Facebook posten.

HTTPS oder nicht HTTPS – das ist die Frage

Wer nun noch automatisch testen möchte, ob Facebook HTTPS auch wirklich eingeschaltet hat – was notwendig ist, wenn der Test anschließend weiterlaufen soll –, muss noch einen kleinen Test schreiben, ob Facebook den Nutzer wieder ausgeloggt hat. Abhängig von dessen Ausgang muss sich das Skript gegebenenfalls erneut einloggen und nochmals prüfen. Praktischeweise bietet die Testplan-Sprache die Möglichkeit, externe Testmodule aufzurufen. Es reicht also, einmal den HTTPS-Test zu implementieren – das eigene Testskript ruft das Modul dann an mehreren Stellen auf. Das erleichtert die Pflege des Tests.

Grundsätzlich zeigt sich die Testplan-eigene Sprache mächtig und als leicht zu erlernen, ist aber schlecht dokumentiert. Das macht anfangs einiges an Experimentieren nötig, bis ein Test wunschgemäß abläuft.

Wiederholung – Täter?

Häufige Versuche in Form von Listing 2 nimmt Facebook übrigens übel und verhängt beim Login einen Captcha – auch die Gegenseite testet also. Testplan einzusetzen lohnt sich wegen seiner Flexibilität übrigens nicht nur für eigene Webanwendungen, sondern auch für kleine (natürlich legale) Angriffe.

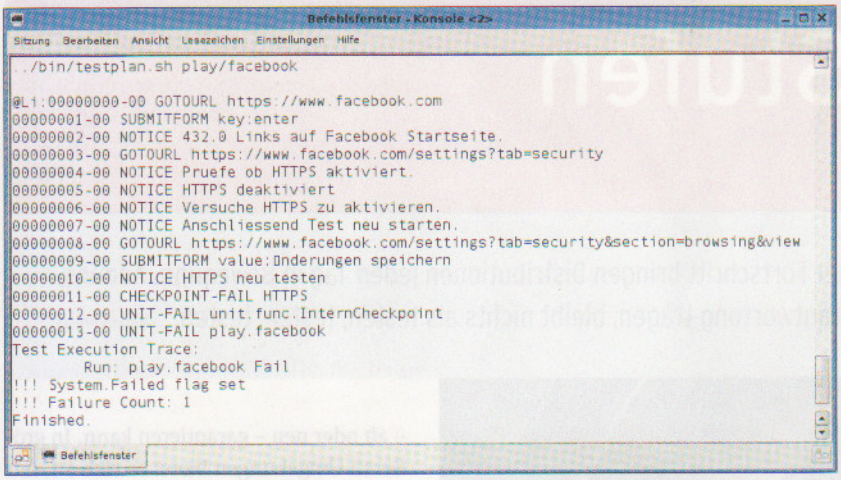

Abbildung 4: Dieser Lauf ergibt, dass HTTPS nicht eingeschaltet ist. Das Programm versucht daraufhin, das Protokoll selbst anzuschalten.

Reaktion auf bestimmte Eingaben. Dazu verfügt es über eine eigene Skriptsprache, die dafür optimiert ist, Grundinteraktionen auf Webseiten zu tätigen, etwa Daten in Formulare eintragen oder Links anklicken. Die Ausgabe kann der Programmierer dann nach Schlagwörtern oder Zeichenfolgen durchsucht lassen. Damit kann er leicht prüfen, ob die Anwendung aus Nutzersicht funktioniert. Wer die Eingabewerten geschickt wählt, kann auch kritische Fälle abfangen (siehe Kasten „Testplan in der Praxis").

Allen Verfahren ist gemein, dass der Entwickler sich darüber klar sein muss, unter welchen Rahmenbedingungen das Programm korrekt abläuft, wo die Grenzen liegen und welche Eingaben es nicht toleriert. Dieses Wissen wandert in die Tests – und wird dadurch explizit. Das reduziert die Fehlerwahrscheinlichkeit.

Lohnt das alles?

Webapplikationen systematisch zu testen ist leider (noch?) zu wenig verbreitet. Kosten- und Zeitdruck seien die Gründe, so ist oft zu hören. Das ist seltsam, denn sobald Angreifer den eigene Online-Auftritt oder den Webshop übernommen haben, fragt niemand mehr nach Zeit und Budgets. Selbst wer im Vorfeld den Verlust an Umsatz und Reputation nicht einrechnen will, sollte bedenken, dass, sobald geeignete Testprozesse etabliert sind, mittelfristig der Mehraufwand überschaubar bleibt. Und wer systematisch testet, prüft die Qualität seiner Arbeit. Das Mehr an Erkenntnis vermeidet künftig gleichartige Fehler und die Qualität späterer Arbeitsergebnisse steigt. *(jk)* ■

Der Autor

Tobias Eggendorfer ist Professor für IT-Forensik in Hamburg. Aus seiner Arbeit mit kompromittierten Systemen kennt er die Folgen schlechter Programmierung nur zu gut. Für Qualität kämpft er zurzeit noch an ganz anderer Stelle: Mit seinem Blog [http://www.meilenschwund.de] will er die Lufthansa daran erinnern, dass auch Kundenbindung Qualität braucht.

Infos

[1] Tobias Eggendorfer, „SQL-Injection legt Newsletter lahm": Linux-Magazin 02/11, S. 108
[2] Tobias Eggendorfer, diverse Artikel über Schwachstellen in Webapplikationen: Linux-Magazin 12/08, S. 78; 01/09, S. 100; 12/10, S. 100; 01/12, S. 48
[3] PHP Unit: [http://www.phpunit.de]
[4] Simple Test: [http://simpletest.org]
[5] Testplan: [http://testplan.brainbrain.net]
[6] PHP-Unit-Handbuch: [http://www.phpunit.de/manual/3.7/en/index.html]

Listing 1: Links zählen

```
01 default %Cmds.Site% http://www.eggendorfer.info/
02 GotoURL %Cmds.Site%
03
04 set %Count% 1
05 foreach %Link% in %Response://a%
06   set %URL% as selectIn %Link% @href
07   Notice %Count% Link: %Link% %URL%
08   set %Count% as binOp %Count% + 1
09 end
```

Listing 2: Facebook testen

```
01 default %Cmds.Site% https://www.facebook.com
02 GotoURL %Cmds.Site%
03
04 SubmitForm with
05   %Form% id:login_form
06   %Params:email% someone@somewhere.com
07   %Params:pass% somesecurepassword
08   %Submit% key:enter
09 end
10
11 set %Count% 1
12 foreach %Link% in %Response://a%
13   set %URL% as selectIn %Link% @href
14   set %Count% as binOp %Count% + 1
15 end
16
17 Notice %Count% Links auf Facebook Startseite.
18 GotoURL https://www.facebook.com/
   settings?tab=security
19
20 Notice Pruefe ob HTTPS aktiviert.
21 Check //span[contains(text(),'Das sichere
   Durchstöbern ist derzeit')]
22 set %alles% %Response://span[contains(text(),
   'Das sichere Durchstöbern ist derzeit')]%
23 if strMatches %alles% ^(Das sichere
   Durchstöbern ist derzeit).* (aktiviert)\.$
24   Pass HTTPS aktiviert
25 else
26   Notice HTTPS deaktiviert
27   Notice Versuche HTTPS zu aktivieren.
28   Notice Anschliessend Test neu starten.
29   GotoURL https://www.facebook.com/settings?
   tab=security&section=browsing&view
30   set %id% %Response://form[@action=
   '/ajax/settings/security/browsing.php']/@id%
31   set %fb_dtsg% %Response://form[@action=
   '/ajax/settings/security/browsing.php']
   /input[@name='fb_dtsg']/@value%
32   SubmitForm with
33     %Form% id:%id%
34     %Params:secure_browsing% 1
35     %Params:fb_dtsg% %fb_dtsg%
36     %Submit% value:Änderungen speichern
37   end
38   Notice HTTPS neu testen
39   Fail HTTPS
40 end
```