网络与计算机应用杂志》109（2018）89-109

内容列表可在ScienceDirect上找到

网络与计算机应用杂志

杂志主页：www.elsevier.com/locate/jnca

DetLogic:检测网络应用程序中逻辑漏洞的黑盒方法

G. Deepa *, P. Santhi Thilagam, Amit Praseed, Alwyn R. Pais

印度苏拉特卡，卡纳塔克国家技术学院计算机科学与工程系

我的朋友们，你们好吗？

关键词。
应用逻辑漏洞 逻辑攻击
网络应用程序安全 参数
篡改 状态违规 工作流违
规 序列违规 授权绕过
认证绕过

ABSTRACT

网络应用程序会受到恶意用户的攻击，这是因为这些程序是由不具备安全编程知识的软件开发人员实施的。由于不安全的编码方式而产生的实施缺陷，使攻击者可以利用应用程序来执行不利的行动，从而导致不良后果。这些缺陷可以被分为注入和逻辑缺陷。由于大量的工具和解决方案可用于解决注入漏洞，攻击者的重点正转向利用逻辑漏洞。逻辑漏洞允许攻击者违背利益相关者的期望，破坏应用程序的特定功能，因此，识别这些漏洞以避免利用是很重要的。因此，我们开发了一个名为**DetLogic的**原型，用于检测不同类型的逻辑漏洞，如网络应用中的参数操作、访问控制和工作流绕过漏洞。DetLogic采用黑匣子方法，将应用程序的预期行为建模为注释过的有限状态机，随后用于推导与输入参数、访问控制和工作流有关的约束。得出的约束条件被违反，用于模拟攻击载体以识别漏洞。DetLogic针对基准应用程序进行了评估，发现其工作效果很好。

1. 简介

网络应用程序在每个人的日常活动中发挥着举足轻重的作用，因为它们广泛存在，并为执行各种活动提供了平台，如通信、银行、购物和社交网络。不幸的是，网络应用的日益普及、可用性、易用性和用户群使其成为网络犯罪分子的诱人目标。网络犯罪分子利用应用程序中的漏洞获取金钱利益，窃取机密信息，进行不受欢迎的操作，等等。根据赛门铁克互联网安全威胁报告（赛门铁克，2016年4月），75%的合法应用程序有未修补的漏洞，2015年有100万次针对网络应用的攻击报告。来自Trustwave（2016）的报告指出，Trustwave测试的97%的应用程序有安全漏洞，每个应用程序发现的漏洞数量中位数为14个。

网络应用程序中的漏洞主要可以分为以下几类两类。注入漏洞和业务逻辑漏洞（Cova等人，2007；Felmetsger等人，2010；Li和Xue，2013，2014）。前者允许攻击者注入恶意数据作为篡改指定结构的命令/查询/脚本，而后者则允许攻击者注入数据，诱使网络应用程序软件表现出不同的行为，并表现出与程序员意图相反的不可预见的行为。大量的解决方案可以用来缓解注入漏洞，因此攻击者的重点已经转移到利用逻辑漏洞上。逻辑漏洞被认为是需要研究人员关注的第二大漏洞（Trustwave，2014）。Trust-wave（2016）的一份报告指出，64%的受测应用程序存在会话管理漏洞，41%存在服务器端漏洞，39%存在认证和授权漏洞。

业务逻辑漏洞是通常允许恶意用户规避应用程序的预期功能的缺陷。利用这些缺陷的攻击是合法的应用交易，用于执行不属于通常商业惯例的不良操作（Grossman，2007）。由于逻辑攻击与经济损失直接相关，逻辑漏洞的影响往往更为严重。例如，考虑一个在线交易应用程序，它为应用程序的每个用户设置了一个交易上限，并通过HTTP请求中的一个参数控制该值。如果该值

* 通讯作者。

*电子邮件地址：*gdeepabalu@gmail.com（G. Deepa），santhi@nitk.ac.in（P.S. Thilagam），amitpraseed@gmail.com（A. Praseed），alwyn@nitk.ac.in（A.R. Pais）。

https://doi.org/10.1016/j.jnca.2018.01.008
2017年9月7日收到；2017年12月25日收到修订版；2018年1月19日接受
可于2018年2月22日在线查阅

如果交易的上限在服务器端没有得到正确的验证，那么任何有知识的用户都可以在请求中自行修改上限，从而违反了应用程序的预期行为。

逻辑漏洞可分为三种主要类型：参数操纵、访问控制和工作流绕过漏洞（Cova等人，2007）。这些漏洞是由于在不同网页之间的数据流动和应用程序的控制流动的管理错误而产生的（Pellegrino和Balzarotti，2014；Monshizadeh等人，2016）。逻辑漏洞与网络应用程序的功能有关，因此识别和处理这些漏洞极具挑战性。

最先进的技术。现有的文献库中有一些解决方案，以确保网络应用在软件开发生命周期的不同阶段免受逻辑漏洞的影响。(i) 对于正在建设中的应用程序，提出了安全的网络框架，以照顾到应用程序本身开发过程中的安全问题（Chong等人，2007b，a；Jia等人，2008；Swamy等人，2008；Vikram等人，2009；Yip等人，2009；Corcoran等人，2009；Swamy等人，2010；Morgenstern和Licata，2010；Krishnamurthy等，2010）。(ii) 对于遗留应用（即已部署的应用），现有的工作利用白盒和黑盒方法来减轻逻辑漏洞/攻击。一些工作集中在检测应用程序中的逻辑漏洞，但把消除这些漏洞的责任放在开发者身上（Bisht等人，2011；Monshizadeh等人，2014；Sun等人，2011；Son等人，2011；Li和Xue，2013；Li等人，2014）。另一方面，一些研究工作侧重于防止应用程序在运行期间的逻辑攻击（Li和Xue，2011；Li等人，2012；Skrupsky等人，2013）。漏洞检测系统帮助开发者识别可利用的弱点，以便对其进行修复，而攻击预防系统则阻止恶意请求绕过应用程序的预期行为。

研究差距。提出的方法/解决方案有其
其自身的优势和劣势描述如下。安全框架不能保护遗留的应用程序免受攻击，而且安全编码做法是劳动密集型的。采用白盒方法的漏洞检测系统需要应用程序的源代码，并且依赖于技术。将应用的源代码分享给第三方来识别漏洞，会损害应用的利益相关者的隐私。攻击预防系统的问题是产生误报，阻止合法交易的执行，以及分析HTTP数据包的源代码工具。漏洞扫描器Acunetix、AppScan、AppScaner、WebInspect和QualysGuard产生传统的攻击载体，如SQL注入、XSS和跨站请求伪造（CSRF），并不涉及逻辑漏洞的识别。因此，需要一个能够检测网络应用程序中的逻辑漏洞的系统，该系统与应用程序的功能无关。因此，我们提出了一个漏洞检测系统，它采用黑箱方法来检测逻辑漏洞。

逻辑漏洞使攻击者能够破坏预期的
应用的行为，因此，逻辑漏洞的识别需要提取应用的预期行为。预期行为被用来生成恶意请求，以帮助检测漏洞。现有的方法考虑到了流向网页的参数、与网页相关的会话变量以及用于维持操作顺序的会话变量，以模拟应用程序的预期行为，从而识别不同类型的逻辑漏洞。现有的方法有其自身的局限性，现讨论如下。(i) 检测参数操作的黑盒方法考虑的是出现在一个网页请求中的参数，而没有考虑到多个网页之间的互动。(即数据流），因此无法识别少数漏洞（Bisht等人，2010）。(ii) 关于工作流旁路漏洞识别的文献考虑了用于维护操作序列的会话变量，而没有考虑CSRF令牌等其他参数（Li and Xue, 2013）。应用程序中的操作序列被称为工作流或控制流。(iii) 识别访问控制漏洞的工作是由于会话变量的不恰当定义和验证，但不够灵活，无法识别其他类型的会话相关漏洞，导致会话困惑和会话修改攻击。(iv) Bisht等人（2010）、Li和Xue（2013）以及Li等人（2014）的工作是临时性的方法，因此无法涵盖更广泛的逻辑漏洞。考虑到最先进系统的优点和缺点，这项工作构建了一个网络应用程序模型，反映了数据流和控制流方面的预期行为，可用于生成攻击请求/载体，以检测所有三种类型的逻辑漏洞。

挑战。识别逻辑脆弱性所涉及的困难
在不使用源代码的情况下，如何提取应用程序的预期行为是一个挑战。本文所涉及的主要挑战和为解决这些挑战而采用的机制如下。

- 了解对应用程序用户输入的业务要求，以确定客户端和服务器端验证之间存在的不一致之处。
  - 对用户输入的要求是通过分析客户端的HTML/JavaScript代码推断出来的。
- 得出应用程序的预期访问控制策略
  - 与应用程序相关的访问控制策略来自为维护应用程序状态而定义的会话变量。
- 推断应用程序的业务工作流程
  - 应用程序中的预期工作流程来自于一个模型，该模型由应用程序中手动完成的导航构建而成。

贡献。在这项工作中，我们开发了一个名为DetLogic的原型，用于识别网络应用程序中存在的业务逻辑漏洞。该原型分三个阶段工作。(i)提取被测网络应用程序的预期行为，(ii)根据收集到的信息构建具体的攻击载体，以及(iii)比较正常执行和攻击执行期间获得的响应，并相应地报告漏洞情况。该原型检测三种类型的逻辑漏洞：参数操纵、访问控制和工作流绕过漏洞。

对这项工作的主要贡献如下。

- 开发一种黑箱方法，提取网络应用程序的数据流和控制流，以识别三种不同类型的逻辑漏洞。
- 构建一个模型，使用提取的信息来反映应用程序的预期行为。
- 从模型中推导出约束条件，并生成违反推导出的约束条件的攻击向量，以识别不同类型的逻辑漏洞。
- 识别导致会话困惑攻击的漏洞。据我们所知，这是第一个识别会话困惑漏洞的工作。
- 在所提出的方法的基础上实现了一个原型，并使用基准网络应用程序评估了原型的有效性。

本文的其余部分组织如下。第二节概述了不同类型的应用逻辑漏洞。

能力及其根本原因。第3节用一个例子说明了这个问题。第4节介绍了拟议的方法和拟议的网络应用模型的概况。第5节和第6节分别阐述了拟议方法的设计和实现。第7节讨论了评估结果。第8节介绍了相关工作，最后一节是本文的结论。

2. 逻辑漏洞

逻辑漏洞允许恶意用户注入数据，诱使网络应用软件表现出不同的行为，并表现出与程序员的意图相悖的未预见的行为，从而导致攻击。

2.1. 逻辑漏洞的类型

三种主要的逻辑漏洞被描述为以下几种。

参数操纵。这些漏洞允许修改关键变量的值，有利于攻击者绕过客户端的估值，对应用程序造成严重破坏。例如，在电子商务应用中，如果在下订单时未能在服务器端对物品的价格进行估值，攻击者就可以将物品的价格修改为负值，从而使攻击者可以免费购买该物品。

访问控制漏洞。这些漏洞允许攻击者获得对受限资源的访问权，该资源是专门为应用程序的高权限用户准备的。 根据OWASP（Top10，2013）风险和SANS错误（SANS，2011），10个风险中的4个[1]和25个危险错误中的5个[2]都与访问控制检查的不正确实施有关。

工作流程绕过。这些漏洞允许攻击者破坏应用程序的预期工作流程，从而破坏应用程序的业务特定功能。预定工作流程是指在应用程序中完成特定任务所需遵循的步骤序列。例如，一个网上银行应用程序要求用户选择一个用于转移资金的受益人，输入要转移的金额，然后确认转移。如果 "转账金额 "网页不检查用户是否已经选择了一个受益人，只访问 "选择受益人 "网页，那么 "选择受益人 "网页就有可能被用户通过篡改 "转账金额 "网页请求中的账户号码，从而形成工作流绕过攻击（Cova等人，2007；Skrupsky等人，2013；Li和Xue，2011）。这使得攻击者可以将资金转移到他们自己的账户。

2.2. 根本原因

逻辑漏洞是由于以下实施缺陷引起的。(i) 缺少服务器端验证，(ii) 缺少和不完整的访问检查，(iii) 会话变量的重载，以及(iv) 缺少序列检查，这些将在下面详细讨论。

缺少服务器端的验证。一个网络应用程序使用客户端脚本来处理和验证用户提供的输入，以便快速处理和降低服务器端的负载。然而，恶意的用户可以通过禁用客户端验证来规避客户端验证。

如果是通过提交恶意请求来篡改参数，从而违反了在客户端对用户输入的限制，那么就会被拒绝。如果应用程序在服务器端执行了同样的限制，那么这些被篡改的参数将被应用程序拒绝。否则，这些参数会导致应用程序的行为与应用程序的要求规格不一致。因此，未能在服务器端执行用户输入验证导致*参数操纵攻击*（Bisht等人，2010，2011；Skrupsky等人，2013）。

遗漏/不完整的访问检查。网络应用程序使用HTTP，这是一个无状态协议，它以独立的方式处理每个网络请求和响应。因此，为了维护状态，应用程序使用会话来表示用户的登录状态、权限级别（例如，管理员或普通用户）、操作顺序以及其他一些事情。会话可以通过两种方式进行维护--纯粹在客户端通过cookies进行维护，或者结合服务器和客户端进行维护。为了维护会话，预计应用程序在允许访问其特权资源之前会对会话变量进行一些检查。例如，应用程序中的基本访问检查会验证会话变量是否被设置，以允许用户在登录后访问应用程序的主页。然而，有些页面可能不执行这些检查，因此，恶意用户可以获得这些页面的访问权。因此，缺失的访问检查会导致*认证/授权绕过攻击*（Sun等人，2011；Son等人，2011，2013；Li和Xue，2011）。

同样地，对于一个用户来说，要访问应用程序，会话变量应该根据用户的角色和HTTP参数进行正确的验证。可能存在这样的情况：应用程序验证会话变量或HTTP参数是否被设置，但没有根据用户的角色和HTTP参数来验证会话变量的值。会话变量对角色和HTTP参数的不恰当验证被称为不完整/不恰当的访问检查，导致*纵向/横向的权限升级攻击*（Li和Xue，2011；Monshizadeh等人，2014）。

会话变量的**重载。**不受控制地创建/填充会话对象或在不同的应用程序入口点使用相同的会话变量被称为会话变量的重载，并可能导致*会话困惑的攻击*（Chen，2011）。这些攻击不包含任何恶意的输入。它们是网络应用程序允许的合法行为，但当以特定的顺序执行时，会承诺应用程序的预期功能。在利用会话拼图时，可以间接地启动会话对象的创建，然后通过以一定的顺序访问一系列的入口点（网页、网络服务、远程过程调用等）来加以利用。会话谜题使攻击者能够执行各种恶意行为，如绕过认证/授权和提升用户的权限，并破坏应用程序的正常执行。

缺少序列检查。网络应用程序使用会话来主-保持应用程序中的操作顺序（Li and Xue, 2011）。除了会话变量，在应用程序的敏感/关键页面生成的CSRF令牌也有助于维护操作顺序（Jovanovic等人，2006）。CSRF令牌存储在会话中或作为一个cookie，并根据HTTP参数进行验证，以防止攻击（CSRF）。CSRF令牌是在应用程序的服务器端在关键页面上生成的，它要么存储在会话中，要么发给客户端并设置为cookie。存储在cookie/会话中的CSRF令牌应该与关键页面之后的网页的HTTP请求中的CSRF参数进行验证。因此，CSRF令牌能够保持应用程序中的操作顺序。如果紧随关键页面的网页未能根据CSRF参数验证存储在会话/cookie中的CSRF令牌的值，那么就会出现以下情况

[1] (1) 破碎的认证和会话管理，(2) 不安全的直接对象引用，(3) 缺少功能级访问控制，以及(4) 无效的重定向和转发。

[2] CWE-306, CWE-862, CWE-250, CWE-863, 和CWE-732。

HTTP参数，那么就有可能被攻击者绕过关键页面，直接向关键页面之后的页面发出请求。因此，在服务器端缺乏对维护序列的令牌的验证，会导致*工作流绕过攻击*。

3. 问题图示

本节以一个运行中的例子来描述这个问题，其中详细介绍了导致网络应用中各种逻辑攻击的各种类型的漏洞。

3.1. 激励性的例子

表1至表8展示了一个网络应用程序样本，用于维护在该程序中注册的病人的医疗记录。该应用程序有六个网页。*登录*、*忘记密码*和*索引*页面是应用程序的所有用户共同使用的，而*查看*、*创建*和*删除*页面是为具有特定角色的用户准备的。应用程序中存在三种用户角色。*病人*、*医生*和*管理员*。该应用程序的业务要求规定了以下规则。(i)病人可以查看自己的记录，(ii)医生可以查看自己病人的记录，并可以创建新的医疗记录，(iii)管理员可以查看和删除任何在应用程序中注册的病人的记录，并可以创建新的记录。

用户首先看到的是*Login.php*页面。在用户
如果用户已经提供了有效的凭证，应用程序会将用户重定向到*Index.php*页面。*Index.php*页面有三个超链接，分别是*View.php*、*Create.php*和*Delete.php*，*这*取决于从数据库中获取的登录用户的角色。*View.php*适用于所有三种角色，*Create.php*适用于*医生*和*管理员*，*Delete.php*只适用于*管理员*.清单3是一个JavaScript文件，用于验证用户在*Create.php*页面提供的输入。

网络应用程序维护两个会话变量--*角色*和*用户ID*。*角色*变量记录了用户在应用程序中的权限，而*userid*变量则通过用户的ID来记录个人用户。 所介绍的应用继承了几个漏洞，这些漏洞将被详细讨论。

```php
<?php
  if(isset($_GET["userid"])) {
    $user = _GET["userid"];
    //Code to create hyperlink to View Record
    page
    .....
    if($_SESSION["userid"]==$user && ($_SESSION["
    role"]=="physician" || $_SESSION["role"]=="
    admin")) {
      //Code to create hyperlink for Create
    Record page
      .....
    }
    if($_SESSION["role"]=="admin" ) {
      //Code to create hyperlink to Delete Record
     page
      .....
    }
  } else {
    throw new Exception("Please login to the
    application");
  }
?>
```

清单1. Index.php。

错误1：*Index.php*页面没有验证会话变量*userid*是否被设置。 因此，任何恶意用户都可以通过在HTTP请求中为参数*userid*提供一个有效的值来访问*Index.php*页面，而无需登录到应用程序中，因此，用户将被提供超链接来查看医疗记录。 因此，利用这种缺失的访问检查导致认证绕过攻击。

```php
<?php
  if(isset($_GET["userid"])) {
    $user = _GET["userid"];
    if(isset( _SESSION["userid"]) {
      if($_SESSION["role"]=="patient") {
      //Code to Retrieve and Display Record
      .....
      } else if($_SESSION["role"]=="physician" ||
     $_SESSION["role"]=="admin") {
        if(isset( _POST["patientid"])) {
          $patientid = _POST["patientid"];
          //Code to Retrive and Display Record
          .....
        } else {
?>
          //HTML Code to get the Patient ID
          .....
<?      }
      }
    }
  } else {
    die("Please login to the application");
  }
?>
```

清单2. View.php。

错误2：*View.php*页面检查*$_SESSION["userid"]*是否设置，但没有用会话变量验证请求中参数*userid*的值。 因此，一个用户可以在登录应用程序后通过修改请求中的*userid*值来查看另一个用户的医疗记录。这种不恰当的页面访问检查会导致横向的权限升级攻击。该页面应该检查传递的*userid*是否与设置的会话变量*userid*相同。

```javascript
function validateInput() {
  var age = document.getElementById("age").value;
  if(document.getElementById("fname").value=="")
    {
    alert("User name can not be blank");
    return false;
  } else if (age < 0 || age > 150) {
    alert("Age must be greater than zero and less
     than 150.");
    return false;
  }
  return true;
}
```

清单3. Validate.js。

```
<script type='text/javascript' src='Validate.js'>
</script>
<?php
  if(isset($_GET["userid"])) {
    $user = _GET["userid"];
    if($_SESSION["userid"]==$user) {
      if($_POST["mode"]=="insert") {
        //Code to Insert Record
        .....
      } else {
?>
        //HTML Code to get inputs for creating record for a user
        .....
<?    }
    }
  } else {
    die("Please login to the application");
  }
?>
```

清单4.创建.php。

错误3：*Create.php*页面检查会话中设置的*userid*和请求中传递的*userid*是否匹配，但未能验证访问该网页的用户的角色。因此，一个拥有较低权限的用户可以通过向*Create.php*页面发出直接请求来创建医疗记录。这种不完整的网页访问检查将导致垂直权限升级/授权绕过攻击。这可以通过检查$_*SESSION*["*角色*"]变量的值是否等于*管理员*或*医生来消除*。

错误4：网页*Create.php*从医生或管理员那里获得输入，以创建医疗记录。在表格中存在一个*年龄*字段，该字段使用清单3中的客户端脚本在0和150之间定义其值。恶意用户可以绕过客户端验证，提交一个*创建*请求。*php*页面，参数*Age*为无效值，例如-5。在将参数*Age*插入数据库之前，服务器端没有对其进行验证，因此违反了应用程序的预期行为。因此，缺失的服务器端验证被利用来发动参数操纵攻击。

```
<?php
  if(isset($_GET["userid"])) {
    $user = _GET["userid"];
    if($_SESSION["userid"]==$user && $_SESSION["role"]=="admin") {
      if(isset( _POST["patientid"])) {
        HTTPRedirect("Confirm.php");
      } else {
        if(!isset($_COOKIE["csrf_token"])) {
          $token = bin2hex(random_bytes(32));
          setcookie("csrf_token",$token);
        }
?>
        //HTML Code to get Patient ID
        .....
<?    }
    }
  } else {
    die("Please login to the application");
  }
?>
```

清单5. Delete.php。

```
<?php
  if(isset($_GET["userid"])) {
    $user = _GET["userid"];
    if($_SESSION["userid"]==$user) {
      if($_POST["confirm"]=="yes") {
        if(isset( _POST["token"])) {
          //Code to Delete Record
          .....
        }
      } else {
?>
          //HTML Code to get Confirmation from user
          .....
<?    }
  } else {
    die("Please login to the application");
  }
?>
```

清单6. Confirm.php。

错误5：*Confirm.php*页面从请求中获得了*用户ID*，但未能验证用户的角色和会话变量的值。因此，任何具有较低权限的用户都可以通过向*Confirm.php*页面发出直接请求来删除现有的医疗记录。这个页面错过了访问检查，因此会导致垂直特权升级/授权绕过攻击。

错误6：网页*Confirm.php*验证了HTTP请求中是否存在CSRF令牌，但未能根据HTTP参数*令牌*验证cookie中设置的CSRF令牌的值。尽管CSRF令牌是用来防止CSRF攻击的，但它间接地帮助维护应用程序中的操作顺序。因此，任何用户都可以通过直接向*Confirm.php*页面发出请求来删除现有的医疗记录，只需在请求中附加参数*令牌即可*。因此，这个网页错过了CSRF令牌的验证，因此会导致工作流绕过攻击。

```
<?php
  if(isset($_POST["userid"])) {
    $_SESSION["userid"] = $_POST["userid"];
    header("Location:PwdRecovery.php?userid=".$_SESSION["userid"]);
  } else {
?>
    <form action="ForgotPwd.php" id="Pwd" method="post">
      UserID: <input type="text" name="userid"><br>
      <input type="submit" value="Next">
    </form>
<? } ?>
```

清单7. ForgotPwd.php。

错误7：应用程序在用户登录时定义了一个会话变量*userid*。应用程序中的一些页面会检查会话变量*userid是否*被设置，是否与请求中的*userid*参数相等。现在，在网络应用程序中有一个密码恢复工具。如果用户忘记了他们的密码，网络应用程序会提示他们的用户ID。一旦输入了用户名，应用程序就会继续显示后续的页面，并将用户名的值存储在会话变量*userid*中，这与登录后存储的会话变量相同。在*ForgotPwd.php*页面中设置会话变量是一个简单的错误，但会产生不可预知的后果。例如，在输入userid后，会话变量被设置。因此，一

个恶意的用户可以立即浏览网页应用程序中的所有页面，而这些页面只检查会话变量*userid*。这是一个例子

会话变量重载。如果会话变量的名称不重复，这个错误就很容易避免。

```
1  <?php
2    if(isset($_POST["userid"]) && isset($_POST["
       password"])) {
3      login = validateUser($_POST["userid"],
       $_POST["password"]);
4      if(! login) {
5        HTTPRedirect("Login.php");
6      }
7      $_SESSION["userid"] = $_POST["userid"];
8      //Code to retrieve role of the user from the
       database
9      .....
10     $_SESSION["role"] = $role;
11     header("Location:index.php?userid=".$_SESSION
       ["userid"]);
12   } else {
13 ?>
14     //HTML form to get login credentials from a
       user
15     .....
16     .....
17 <? } ?>
```

清单8. Login.php。

Bug 8: *Login. php*从用户那里获得输入，验证输入并将用户重定向到*Index.*如果提供的用户输入无效，那么清单8的第5行将用户重定向到同一个页面，即*Login.php*，但是服务器执行其余的代码（从第7行开始），因为它没有嵌入到else子句中。换句话说，即使提供了无效的凭证，网页仍然设置了会话变量，因为这些变量可以访问应用程序中的所有网页。这种漏洞被称为重定向后执行（EAR）漏洞（Doupé等人，2011；Payet等人，2013；Li和Xue，2013）。这种错误可以通过在适当的区块中正确嵌入源代码来避免。

4. 建议的方法

给定一个网络应用程序，这项工作的目的是开发一种系统的方法，用于对应用程序的预期行为进行建模，以检测应用程序中普遍存在的业务逻辑漏洞，这与应用的功能无关。制定的网络应用程序模型应反映数据流和控制流方面的预期行为，随后用于生成攻击载体以识别漏洞。在这项工作中，数据流是指在不同的网页之间流动的参数和会话变量，而控制流是指应用程序的网页之间的操作序列。

基于所提出的方法实现了一个原型，它构建了一个用于识别逻辑漏洞的网络应用程序行为模型。本节描述了为识别漏洞而构建的原型的结构，以及为反映应用程序的预期行为而提出的概念模型。

4.1. 概述

作为这项工作的一部分，开发了一个名为DetLogic的原型，用于识别三种不同类型的业务逻辑漏洞，即参数操纵、访问控制和工作流绕过漏洞，在网络应用中使用黑盒方法。原型是一个渗透测试工具，如图1所示，在被测网络应用程序的客户端和服务器之间使用。

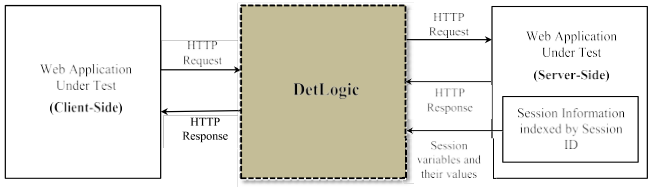

图1. DetLogic概览。

DetLogic作为一个代理，在学习阶段拦截被测程序的HTTP请求和响应，在发现漏洞时向被测程序的Web服务器发出攻击请求。DetLogic的工作原理将在第5节解释。

4.2. 网络应用模式

网络应用的行为在概念上被建模为一个固定的状态机，本节将详细描述。

有限状态机（FSM）被用来为应用程序的行为建模，因为FSM很适合为任何系统的行为建模。构建FSM所需的信息是从使用代理服务器收集的执行痕迹中提取的。代理服务器拦截对网络应用的HTTP请求，并提供构建FSM所需的参数和会话变量的详细信息。

4.2.1. FSM建设

为了识别三种类型的逻辑漏洞，网络应用被建模为具有六元组（$Q, \Sigma, A, \delta, q_0, F$）的注解FSM（$W$），其中

- Q表示有限的状态集，这些状态被表示为应用程序中具有唯一URL的网页的DOM结构。
- Σ表示有限的输入集合。在FSM中标志着过渡到下一个状态的输入符号被表示为被点击的超链接的URL或应用的重定向网页的URL。
- A是一组注释，用于提供附加条件，协助从当前状态过渡到下一个状态。注释被表示为一个三个元组，其内容为 [R，P，S]，其中
  - R代表对该页面有访问权的角色集合（即用户的特权级别）。
  - P代表HTTP参数的集合和它们各自的值，这些参数会随着请求被传递。
  - S代表会话变量的集合和它们的相应值
- δ是过渡函数，它被定义为从$Q\times\Sigma\times A$到Q的映射。例如，$\delta(q_s, i, a) = q_d$表示，如果当前状态是q_s，输入符号i（$i\in\Sigma$）在注释a（$a\in A$）下，那么将有一个从当前状态q_s到下一个状态q的过渡$_d$。我们称这种FSM为注解FSM。
- q_0标志着初始状态，其中$q_0\in Q$。初始状态是应用程序的主页，导航从这里开始。
- F代表最终状态的集合，其中$F\subseteq Q$。最终状态是指标志着应用程序内导航结束的页面（即通常是*注销*页面）。

应用程序中具有唯一URL的网页的DOM结构在注释的FSM中被表示为一种状态。边缘用URL、HTTP参数和由用户和应用程序输入/设置的会话变量来标记。关于构建状态机的信息，在以下章节有详细解释

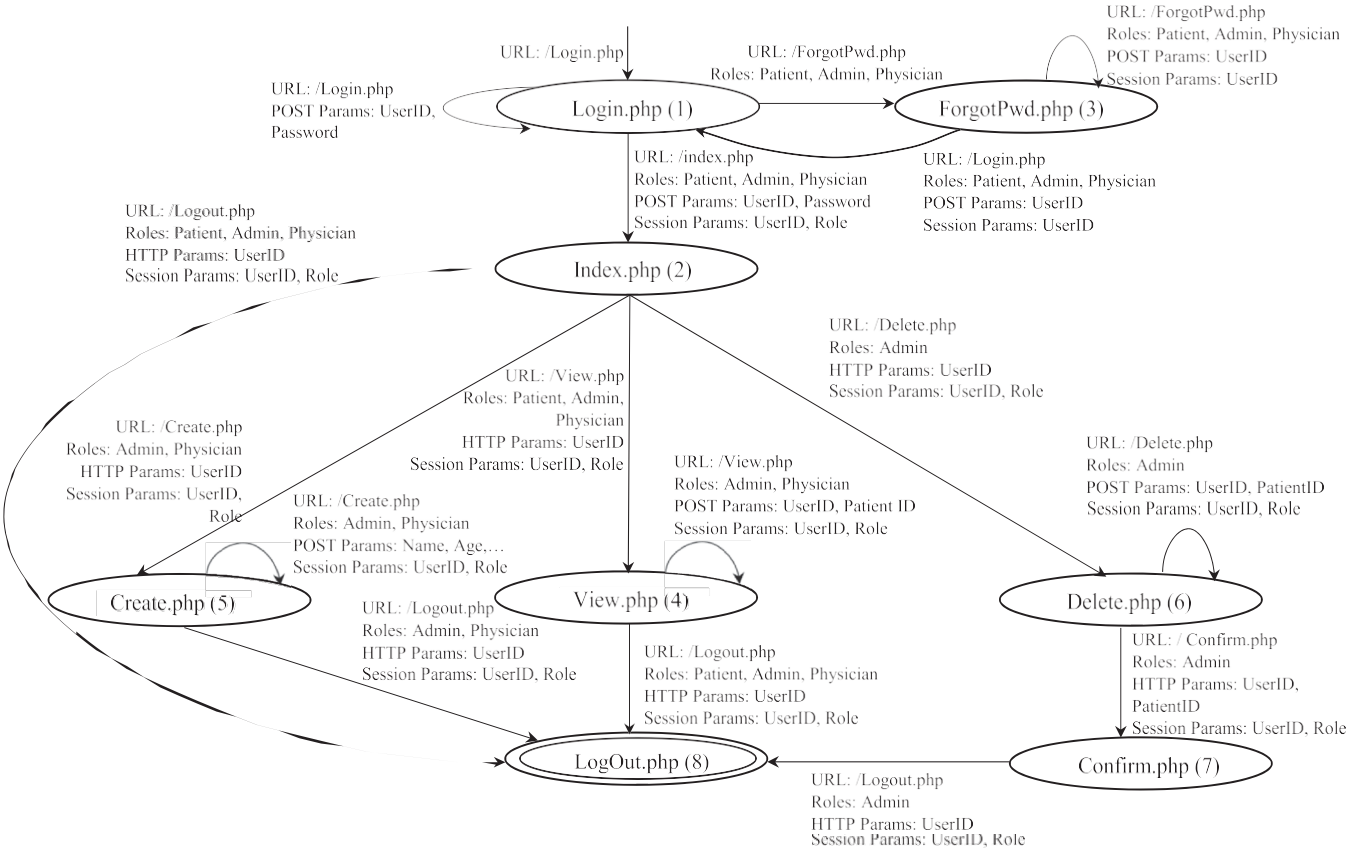

图2.网络应用程序模型。

第6.2.1节。标签边上的URL被视为输入符号，而参数和会话变量被视为注释函数。在一个特定网页上的输入符号和注释函数决定了向后续网页的过渡。在FSM边上标记为注释的参数代表数据流，会话变量及其值用于推断访问控制策略，从一个状态到另一个状态的转换代表应用的控制流。数据流和控制流分别用于识别参数操纵和工作流绕过漏洞，会话变量则用于识别访问控制漏洞。

在一个理想的情况下，如果应用程序中的所有网页都是测试人员对所有角色和所有可能的工作流进行浏览，那么第3节中讨论的激励性例子可以被建模为图2所示。 注释的FSM有八个状态，状态图中边上的输入符号标志着应用程序中的转换。*Login. php*和*Logout. php*分别标志着应用程序的初始和最终状态。该模型反映了应用程序的预期业务行为。*登录*、*索引*、*查看*、*忘记密码*和*注销*页面可以被所有用户访问，而*创建*和*删除*页面分别为管理员、物理学家和管理用户准备。如图2所示，在网页*Index.php*中，如果用户点击了创建新医疗记录的超链接，那么如果会话变量*userid*和*role*被设置，并且HTTP请求中存在参数*userid*，那么就会过渡到下一个状态，*Create.php*。边缘上标记的输入符号*role*代表*Create.php*对用户可用的角色。

4.2.2. 限制条件的推断

生成的网络应用程序模型（注解的FSM）被分析用于推断与参数（即数据）、访问控制（即角色）和应用程序的控制流有关的约束条件，这些约束条件随后被用于生成具体的攻击向量，以识别三种类型的逻辑漏洞。关于提取约束条件的细节将在以下几个小节中详细描述。

*4.2.2.1. 访问控制约束。*构建的状态机被用来概括一套访问控制约束，用于推导攻击载体。三种类型的访问控制约束是由会话变量和HTTP参数以及它们在每个状态下各自的值推断出来的，解释如下。首先，从构建的模型中推断出的每个状态的基本约束是对导致过渡到该状态的会话变量和参数的无效检查。第二种推断的约束是来自会话变量和HTTP参数的平等约束，其值是相同的。第三种推导出来的约束是基于对该状态有访问权的用户的角色。从图2中得出的一些约束如下。

在*Create.php*中推断出的约束条件是。

(i) $_*SESSION*[*userid*] ≠ *null*,
(ii) $_*POST*[*userid*] ≠ *null*。
(iii) $_*SESSION*[*userid*] == $_*POST*[*userid*], 和
(iv) $_*SESSION*[*角色*] == *管理员* 或 *医师*

同样地，*Delete.php*推断出的约束条件包括

(i) $_*SESSION*[*userid*] ≠ *null*,

。

(ii) $_POST[userid] ≠ null。
(iii) $_SESSION[userid] == $_POST[userid], 和
(iv) $_SESSION[role] == Admin

会话变量和检查空值的参数（约束(i)和(ii)）有助于识别缺少访问检查的漏洞。约束(iii)根据HTTP参数的值来检查会话变量的值，这对识别水平权限升级漏洞很有用。会话变量对照角色检查其值（约束(iv)），与应用程序一起注册，有助于识别授权绕过漏洞。攻击向量是通过违反每一个提取的约束条件来产生的，以确定漏洞。为了检查Con- straint (i)是否被执行，*Create. php*页面被强行浏览，会话变量*userid*为空。对于约束条件（iii），HTTP请求中的参数*userid*的值被修改为与会话变量*userid不同的*值。在Con- straint (iv)的情况下，对网页*Create.php*和*Delete.php的*请求是由具有较低权限的用户（即病人）提交的。如果攻击向量得到的响应与正常请求相同，那么这些网页的漏洞将被报告。

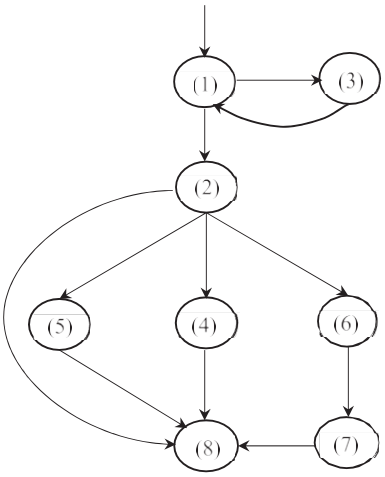

图3.控制流程图。

*4.2.2.2. 参数约束。在*客户端对用户输入的限制被存储为参数相关的约束（Con- straint (ii)）。使用第6.2.2节中讨论的分析器对客户端代码进行分析，以提取这些约束，并映射到模型上。该JavaScript分析器提供了三种不同类型的网页之间流动的参数的约束。(i) 数据类型约束，(ii) 价值约束和(iii) 长度约束。数据类型约束是指用户应该提供的输入值的类型。价值约束指的是一个输入字段所能容纳的价值。长度约束指的是一个输入可以容纳的字符数。例如，对于*Create.php*中的参数*Age*，推断出的价值约束是0 < *age* < 150。同样，对于参数*name*，推断的长度约束是*Length*(*fname*)>0。在攻击生成过程中，推断的约束被违反，以检测参数操作漏洞。如果攻击请求得到的响应与正常请求的响应相似，那么网页在服务器端就缺少对参数的验证。

*4.2.2.3. 工作流限制。*关于工作流绕过漏洞的检测，应用程序中的敏感页面将被识别，以验证它们是否可以被绕过。在这项工作中，敏感/关键页面被定义为网页，其开发目的是为了在工作流中不被跳过。被识别的敏感/关键页面被标记为工作流程约束。正如第2.2节所述，应用程序中的操作顺序是使用会话或CSRF令牌来维护的。识别应用程序中的关键页面有助于揭示那些由于缺乏会话变量或CSRF令牌的验证而无法维持顺序的页面。虽然缺乏对HTTP参数的验证的会话变量也可以使用访问控制约束来识别，但缺乏对存储在cookie中的CSRF令牌的验证的页面只能从应用中发现的关键页面中识别出来。

关键网页可以通过将网络应用程序建模为一个有向图来识别。为了实现这一点，从注释的FSM中构建了一个底层有向图*G*。FSM的状态（即所有网页的集合）代表*G*的顶点集，*G*的边集通过连接两个顶点（即网页）得到，比如*u*和*v*，如果存在从顶点*u*到顶点*v*的工作流，则用一个弧连接。工作流图将是强连接或弱连接的。也许，在大多数情况下，工作流图是弱连接的。工作流图中的关键页面是通过获得*G*的切顶点来识别的。*G*的切顶点是*G*中的一个顶点，它的移除会增加*G*的组件数量或断开*G的*连接。在这项工作的背景下，切顶点指的是一个网页，当跳过它时，会断开/干扰应用程序的工作流。换句话说，工作流图的切顶点对应于网页的开发，其目的是在工作流中不应该被跳过。在这里，工作流图中的切顶集合是通过对图进行深度搜索得到的。另外，如果删除一个顶点不会断开有向图的连接，但在第一深度搜索期间从根顶点创建了未访问的顶点，那么它也被认为是一个关键页面。因此，对应于切顶和上述条件的网页被存储为工作流约束/关键页。涉及这些关键网页的工作流被视为识别工作流绕过漏洞。在一个工作流中跳过对关键网页的请求，以检测关键网页是否可以被绕过。

图3表示从注解的控制流图中获得的控制流图。图2中所示的FSM。图中的切割顶点是节点（1）和（2）。此外，删除顶点（6）使顶点（7）在第一深度搜索期间不被顶点（1）所访问。因此，从应用中推断出的关键页面是*Login*（1）、*Index*（2）和*Delete.php*（6）页面。图4a、b和c表示去除关键页面后得到的图形。

因此，注释的FSM被有效地用于提取参数约束、访问控制约束和工作流约束，这些约束随后被违反，分别用于识别参数操作、访问控制和工作流绕过漏洞。表1描述了第3.1节中讨论的应用中存在的缺陷和推断出的约束。

*该模型的有效性。*所提模型的eff效性说明如下。

- FSM边上的注释使我们能够识别三种不同类型的逻辑漏洞。
- 与LogicScope（Li and Xue, 2013）相比，标注在注解的FSM边缘上的会话变量使该模型能够识别与会话管理相关的新型漏洞，并且具有适应性。诸如会话标识符处理不当的漏洞（例如，在登录前向用户发布会话标识符，但在登录后未能修改这些标识符的页面）可以通过将会话标识符映射到模型上并触发适当的攻击矢量来识别。

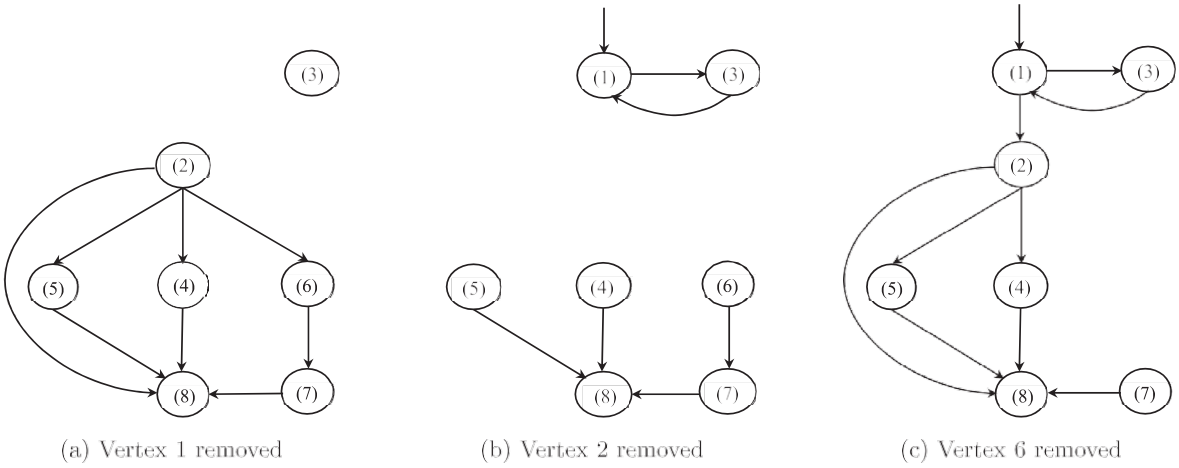

图4.删除关键页面后的图表。

表1
错误和限制。

	上市公司Bug	拘束	地点
清单1	遗漏的访问检查	$_*SESSION*[*userid*] ≠ *null*	4号线
清单1	不完整的访问检查	$_*SESSION*["*userid*"] == $_*POST*["*userid*"]	10号线
清单2	不正确的访问检查	$_*SESSION*["*userid*"] == $_*POST*["*userid*"]	4号线
清单4	不完整的访问检查	$_*SESSION*["*role*"] == "*physician*" ‖ "admin"	6号线
清单6	遗漏的访问检查	$_*SESSION*["*role*"] == "*admin*", $_*SESSION*["*userid*"] == $_POST["userid"]	第四行
清单7	会话变量重载		三号线
清单4	缺失的服务器端验证	*年龄*>0 && *年龄*<150	第9行
清单6	遗漏的序列检查	Delete.php (关键页)	第5行

检察官。

- 注释的FSM模型能够识别由于服务器端缺少CSRF令牌验证而产生的工作流旁路漏洞，这在BLOCK（Li和Xue，2011）和LogicScope中没有得到解决。
- 该模型可以灵活地识别应用程序中存在的漏洞的严重程度。控制流图可用于通过计算节点的外度来标记更关键的节点。识别出的关键节点可以根据它们的出度进行排序，随后可用于标记网页的关键性。
- 尽管与LogicScope相比，注释的FSM的状态数量更多，但在忽略网页中的数据内容后，我们倾向于使用网页模板（即DOM结构）来表示FSM的状态，从而使状态的数量最小化。对于任何大型应用，如电子商务应用，FSM的大小不会随着显示数百万种正在销售的产品的网页数量而增加。由于DOM结构的相似性，所有这些用于显示产品的网页在构建FSM的状态时将被视为一个单一的页面。

5. 拟议的系统结构

DetLogic的原型分三个阶段运行。(i) 学习阶段。(ii)攻击生成阶段，和(iii)发现阶段。学习阶段是为了提取应用程序的预期行为。它提取施加在参数上的约束，并构建一个模型，反映应用程序的预期工作流程。攻击生成阶段是为了生成具体的攻击向量，以帮助识别应用程序中的漏洞。　在识别阶段，该原型试图通过比较在学习阶段获得的响应来检测网络应用的漏洞。

摄取和攻击生成阶段。图5显示了DetLogic的高层系统设计。DetLogic的工作原理详见下文。

输入。种子URL和有效的用户凭证被作为输入，以确定参数操作的注入点。测试人员产生手动跟踪，以确定访问控制和工作流绕过漏洞。

输出。输出是一份漏洞报告，其中列出了逻辑漏洞、利用漏洞可能进行的攻击类型，以及被测网络应用程序中存在的漏洞的位置。

跟踪收集。追踪是在无攻击会话下的应用程序导航过程中产生的HTTP请求。这些痕迹可以手动或自动生成。对这些痕迹进行分析，以提取应用程序的预期行为。这项工作通过手动和自动生成痕迹来推断应用程序的规格。手动追踪用于推断应用程序的控制流。使用代理拦截手动生成的HTTP请求，以提取会话相关信息，这对识别访问控制漏洞非常有用。自动追踪是使用爬虫生成的，爬虫以深度搜索的方式探索应用程序中的所有网页。

学习。学习阶段负责推断出网络应用的预期行为。一个HTML/JavaScript分析器被用来推导参数的约束条件，一个模型生成器被用来以注释FSM的形式提取应用程序的预期行为。在跟踪收集过程中，从提交给应用服务器的HTTP请求中产生的HTTP响应被收集、存储，并提供给JavaScript分析器和模型发生器。分析器和模型生成器处理请求和响应，并分别以参数约束和注解的FSM的形式表示对参数和应用程序内的导航的限制。注释的FSM提供了应用程序的数据流和控制流。

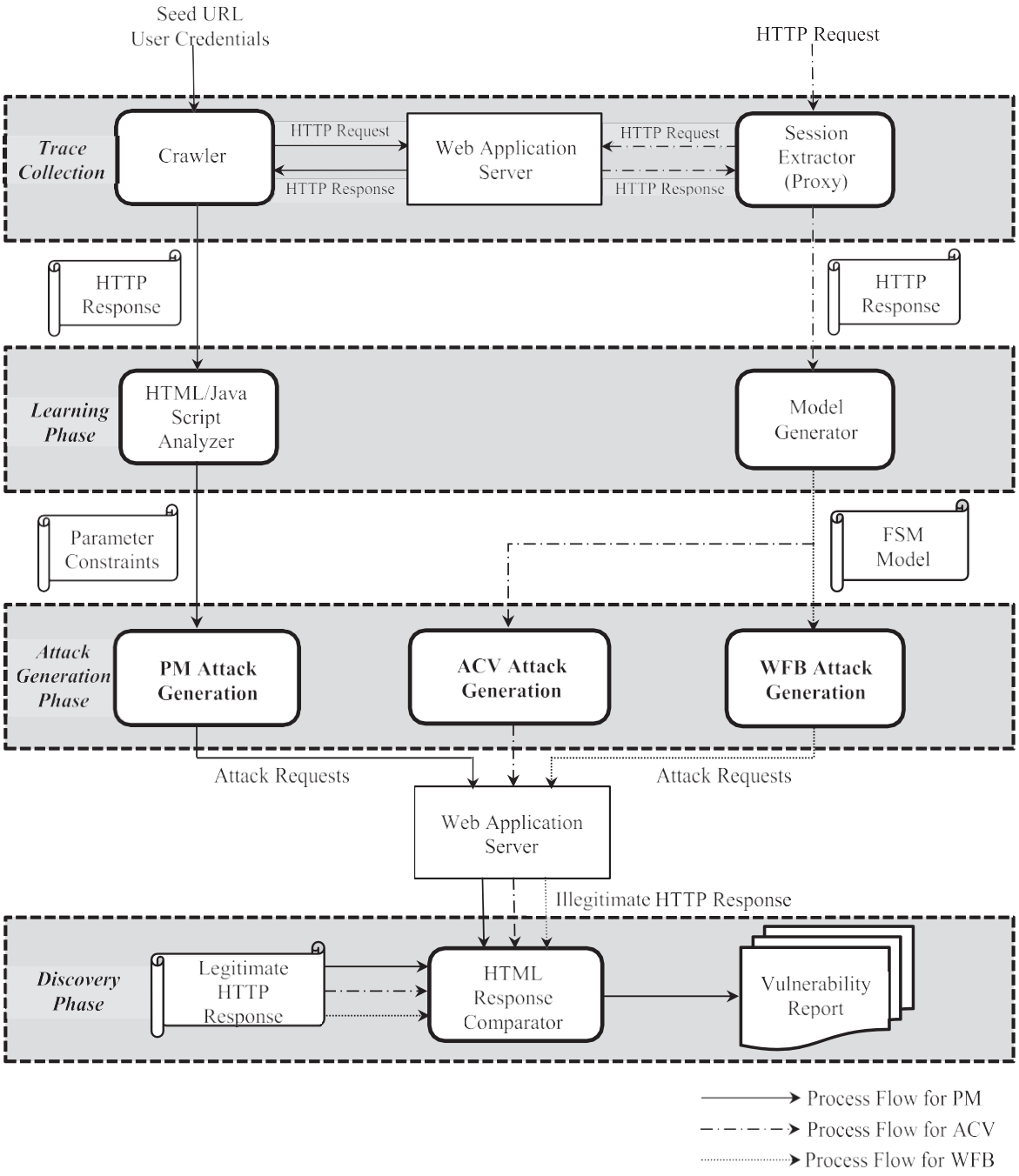

图5. DetLogic架构。

攻击生成。三个不同类型的攻击参数操纵、访问控制违规和工作流绕过攻击被生成，用于识别业务逻辑漏洞。攻击向量是通过修改HTTP请求中的参数值来产生的，这样就违反了参数约束，或者附加了请求中不应该出现的参数。为了识别违反访问控制的攻击，访问控制约束被用来识别高权限的页面是否可以被低权限的用户访问。为了识别工作流绕过攻击，从控制流中识别出网络应用中的关键网页，并跳过其访问。

发现。开发了HTML响应比较器，用于比较每次攻击请求后产生的响应与学习期间存储在数据库中的相关响应。当攻击过程中产生的响应与数据库中的响应相似时，就会报告漏洞。

在一个有效的请求中得到的响应。最后生成一份报告，其中包括漏洞的细节、利用漏洞的攻击类型以及导致攻击的flaws。

6. 实施

本节阐述了原型的实施细节。

6.1. 痕迹收集

追踪是探索被测试的网络应用程序的关键。追踪可以通过爬虫自动生成，也可以由测试人员手动生成。由于以下原因，在检测访问控制和工作流绕过漏洞时，手动跟踪生成比自动爬行更受欢迎。

(i) 有可能将应用程序的错误行为推断为爬虫的预期行为。例如，考虑一个容易受到授权绕过攻击的应用中的一个网页。在使用自动爬虫在学习过程中探索应用程序的情况下，爬虫能够以正常用户的身份访问易受攻击的页面，因此会导致推断出一个错误的条件，即该页面可以被正常用户访问，而不是高度授权的用户。由于这种错误的学习，应用程序中的漏洞就会被错过，无法被识别出来。因此，如果应用程序有漏洞，那么自动抓取就会导致推断出错误的行为，从而产生错误的否定结果。

(ii) 有可能会遗漏一些脆弱的页面，从
由于爬虫可能不会按照程序设计的方式对应用程序进行导航，因此爬虫可能会识别出这些页面。例如，考虑一个电子商务网站，它有以下网页，即*购买*、*添加新商品*、*编辑商品*、*删除商品*，然后是*确认订单*和*支付*页面。假设*"购买"*页面有指向*"添加新商品"*、*"编辑商品"*和*"删除商品"页面的*超链接，如果爬虫在访问*"编辑商品"*或*"删除商品"*网页时，购物车中没有任何商品，那么接下来的*"确认订单"*和*"付款"页面就*有可能无法被完全处理。由于这些网页是在*"编辑物品"*或*"删除物品"*工作流程中访问的，即使通过*"添加新物品"*页面添加了新的物品，工作流程也不会完成，因为爬虫将"确认*订单*"和*"付款"*页面标记为已经访问过。这可能导致对应用程序行为的错误推断。

(iii) 为了识别工作流旁路漏洞，所涉及的主要挑战是提取网络应用程序中的所有有效工作流。这是因为，今天的网络应用程序改变状态不仅仅是由于对超链接的点击操作，而且也是由于嵌入在网页中的元素所触发的事件。抓取这样的应用程序是很困难的，因为爬虫不能启动一个JavaScript事件。因此，这项工作利用了人工生成的痕迹。

人工追踪是通过让测试者在一个被配置为使用代理服务器的浏览器中浏览应用程序而产生的。代理服务器拦截HTTP请求/响应，这些请求/响应被储存起来，以便在构建模型和发现漏洞的过程中进一步分析。为了达到更好的覆盖率，对应用程序的每个角色都进行了探索。除了收集网络跟踪请求外，还提取了对维持应用程序状态至关重要的会话变量。这些变量有助于检测访问控制方面的漏洞。为了实现这一点，代理服务器被配置了一个扩展模块，负责提取会话信息，这一点将在第6.1.2节讨论。

6.1.1. 爬行器

作为这项工作的一部分而开发的爬虫组件从种子URL开始爬行应用程序并获取HTTP响应。爬虫在网页的HTML响应中寻找超链接。它寻找HTML内容中的*src*、*href*和*action*等属性，以及*window.location*、*window.open*、*.location.assign*、*.href*、*.load*、*.action*和*.src*等JavaScript事件来识别网页中的URL。捕获的URL被存储在一个列表中，爬虫开始以深度搜索的方式探索应用程序的网页，直到所有的网页都被看到。爬虫组件的工作原理和算法可以在Palsetia等人（2016）中找到。在不同的网页之间流动的参数被观察和记录。获得的响应被存储在数据库中，以便进一步处理。由于以下原因，爬虫的覆盖属性不能得到保证。

(i) 爬虫不考虑嵌入在主动连接中的链接。像Flash和Silverlight这样的帐篷，并且不考虑AJAX请求、文件、图像等。(ii) 爬虫不支持类型增强的表单提交，即爬虫不考虑应用中对输入字段的限制。因此，对该特定网页的请求可能会被应用服务器忽略，因此，少数网页可能会被忽略。

6.1.2. 会话提取器

会话可以通过两种方式进行维护--纯粹在客户端通过cookie进行维护，或者是服务器和客户端的结合。在后一种方法中，一个包含唯一会话标识符（即会话ID）的cookie被保存在客户端。对于每个请求，这个cookie被发送到服务器，然后服务器使用会话ID与存储在服务器上的会话信息相匹配。会话ID被用来提取实际的会话变量名称和它的值，然后与URL进行映射。PHP应用程序的会话信息提取描述如下。会话信息被存储在服务器端，并以会话ID为索引。会话ID通常与HTTP响应头一起传递。这方面的例外是Ruby，它通过cookies传递整个会话数据。PHP将会话信息存储在一个临时文件夹中。在 Linux 安装中， 它通常存储在 ~~/opt/lamp/temp~~ 文件夹中。会话信息被序列化并以变量名-值对的形式存储在会话文件中。这些信息被反序列化，并为每个请求-响应对提取。

6.2. 学习阶段

6.2.1. 发电机模型

算法1给出了从执行跟踪中构建注释的FSM的伪代码。该算法观察HTTP请求和响应，同时测试人员正在浏览应用程序。每当一个请求被提出，它就被添加到输入符号集（Σ）。例程*ExtractParams()*从请求中获取HTTP参数，并将其存储在参数集（*P*）中。在收到响应后，该算法将验证该网页是否已经被访问过。如果没有被访问过，那么就会创建一个新的状态，并从源状态中标记一个过渡，将与过渡相关的会话变量和参数添加到注释集中。初始状态和源状态在算法执行前被初始化为空值。在算法执行过程中创建的第一个状态被分配给变量q_0。如果网页已经被访问过，那么就用函数*getExistingState()*检索该网页对应的状态，并在源状态和现有状态之间创建一个新的过渡。如果两个状态之间已经存在一个具有不同会话变量的过渡，那么新的会话值将被附加到相应的过渡函数中。创建的新状态或当前状态（即用户当前所处的网页）成为下一个转换的源状态。

使用例程*ExtractSes- sion()*提取会话变量，并添加到注释集（*A*）中的会话变量集（*S*）。该例程在提取会话之前识别了服务器端的技术。如第6.1.2节所述，"会话提取器 "模块将网页URL与会话变量进行映射，并存储以下信息：登录信息、HTTP参数及其值、会话变量及其值，以及HTML响应。

在构建模型后，分析了通往具有相同输入符号的状态的过渡函数，以及HTTP参数和会话变量的不同值，并提取了代表用户角色的会话变量（*S*）的值，存储在变量*SetofAccessibleRoles*中。这就提供了关于能够访问特定状态（即网页）的角色的信息。