NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications

Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V.N. Venkatakrishnan

University of Illinois at Chicago
Web Applications

• Common Characteristics
  • Content generated on the fly to improve usability and responsiveness
  • Tasks require a series of steps to accomplish
    • e.g., online shopping: view → select → add to cart → checkout
    • Dependencies among them
  • However
    • Increase application complexity
    • Increase analysis difficulty
Web Application Example

- **How to Exploit?**
  - Find a vulnerability
  - Craft an exploit string for that vulnerability
  - Find a navigation path to the vulnerability
- **Exploit is:**
Problem & Challenges

- **Problem**: How to automatically construct exploits for large and complex web application?

- **Challenge #1**: Scalability:
  - Large code base consisting of hundreds of modules with large number of intra-module execution paths

- **Challenge #2**: Sinks reachability:
  - Have to derive inputs that reach ‘deep sinks’
  - Exploit input has to
    - navigate through the complex dependencies among modules
    - satisfy module and path constraints
Challenges

• Challenge #3: Dynamic features of web applications
  - dynamically generated content may drive the navigation of the application to vulnerable sinks
    • Forms, links, JavaScript content

• Challenge #4: handling multiple vulnerability classes
  - e.g., injection vulnerabilities (SQLI, XSS, etc.) and logic vulnerabilities (e.g., EAR)
    • minimal changes to the analysis

Goal: Automatic exploit generation approach that addresses these challenges

Our Main Contribution: NAVEX, a system that has identified over two hundred exploits in modern PHP web applications
Approach Overview

• Find vulnerable sinks using static analysis methods
• Build a graph representation of navigation structure of applications dynamically
• Find navigation paths to the identified vulnerabilities
• Final exploit construction
NAVEX Architecture

App source code

Vulnerable Sinks Identification

Vulnerable Sinks
Exploit Strings

Concrete Exploits Generation

Exploits

Attack Dictionary
Step I: Vulnerable Sinks Identification

- **Graph model of source code**
  - Based on Code Property Graphs (CPGs)
    - $\text{CPG} = \text{AST} + \text{CFG} + \text{call graph} + \text{DDG}$
  - Extend CPGs with *sanitization* and *database constraints* tags

- **Find vulnerable paths to sensitive sinks**
  - Path sensitive analysis
  - **Types:** Forward and *backward* traversals based on vulnerability type
    - E.g., *backward* search for injection vulnerabilities

- **Construct formulas from vulnerable path statements**
  - Use solver to generate *exploit strings*
Step II: Concrete Exploits Generation

- **Links**: stored and used as new URLs to crawl
- **Forms**: Generate *form inputs automatically*
  - Extract constraints from forms
- **JavaScript**: concolic execution based on NoTamper (Bisht et al., CCS’10)

- An *application-wide navigation graph*
  - represents possible sequences of module executions
- Directed graph
  - *node*: HTTP request
  - *edge*: navigation between nodes (type is *link* or *form*)

- Search the NG to find navigation paths to vulnerable sinks
Input Generation

Client Formula extraction

\[ F_{\text{form}}: \neg F_{html} \land F_{js} \]

Constraint Solver

\[ F_{\text{form}}: \neg F_{html} \land F_{js} \land F_{server} \]

Solver model

HTTP request Formulation

crawler

Extract Trace

constraints \((F_{server})\)

Trace Analyzer

Store info/create NG Node
Combining Static & Dynamic Results

- **Example:**
  - vulnerability in `PathToApp/App/checkout.php`, checkout.php is included by `hold.php` (no direct access)
  - Navigation Graph: **no node of a URL = “….checkout.php”**

- **Problem:** combining the results produced by the step of vulnerable sink identification (static analysis) with the Navigation Graph (dynamically generated).

- **Solution: Inclusion Map**
  - Constructed statically, [Parent file -> included files]
Searching Navigation Graph

**Input:**
- **vulnerable sink (destination URL)** = http://localhost/App/hold.php
- **exploit string** is msg =<script>alert("XSS");</script> (GET)
- **Public URL (source URL)** = http://localhost/App/selectBooks.php

**Search Results:**
- nodes of [id=2, id=3, id=4, id=5, id=6]
Final Exploit

   POST params: [book name=intro to CS by author1, edition=2,publisher=aaaaaaaa]
EVALUATION
Dataset

- **26 real-world** open-source **PHP** web applications
- Total of **3.2M SLOC** and **22K PHP files**
- Applications selection criteria
  - **Popular and large** PHP apps
  - **Comparison** with **state-of-the-art work** in exploit generation (e.g., Chainsaw (Alhuzali et al., CCS’16)) and vulnerability analysis (e.g., RIPS (Dahse and Holz, NDSS’14))
Results Summary

- NAVEX constructed a total of 204 exploits
  - 195 are on injection vulnerabilities (SQLI and XSS).
  - 9 are on logic vulnerabilities (EAR).
- The enhanced CPG reduced FPs by 87% on average.
- Client-side code analysis for building the navigation graph enhanced the precision of exploit generation by 54% on average.
- Drill down as deep as 6 HTTP requests to stitch together exploits.
SQLI Exploits

- Reported **155 SQLI exploitable sinks**
- No false positives
- Constructed **105 concrete SQLI exploits**
- **Vulnerable** web apps
  - osCommerce (2.3.3)
  - phpBB (2.0.23)
  - myBloggie, Scarf, Dnscript, WeBid, Eve, SchoolMate, geccbblite, FAQforge, and WebChess
XSS Exploits

- Found 133 XSS exploitable sinks
- 5 false positives
- Generated 90 XSS exploits
- Vulnerable web apps
  - HotCRP (2.60)
  - osCommerce (2.3.4)
  - osCommerce (2.3.3)
  - CPG
  - MediaWiki
  - phpBB (2.0.23)
  - myBloggie, Scarf, Dnscript, WeBid, Eve, SchoolMate, FAQforge, and WebChess
EAR EXPLOITS (LOGIC EXPLOITS)

- Found **22 EAR** vulnerabilities
- 3 false positives
- Generated **9 EAR exploits**

**Vulnerable web apps**
- HotCRP (2.100)
- HotCRP (2.60)
- OpenConf
- osCommerce (2.3.4)
- osCommerce (2.3.3)
- LimeSurvey
- Collabtive
- MediaWiki
- myBloggie, WeBid, and Eve
Performance and Scalability

- **Performance**: total time to find exploitable sinks and to generate exploits per vulnerability type.
  - **Vulnerability identification** from 17.28 to 109.27 minutes.
  - **Exploit generation** from 1.38 to 40.20 minutes.
Effect of Client-side Code Analysis

- Forms are common
  - Number of unique forms ranges from 3 to 186 (average of 45 form/application).
- Input generation and constraints extraction from client-side code → improve the crawling coverage.
- NAVEX constructed more exploits.
Conclusion

• NAVEX is an automatic exploit generation system that considers
  • dynamic features and the navigational complexities of modern web applications
• NAVEX constructed 204 exploits
  • 195 are on injection vulnerabilities
  • 9 are on logic vulnerabilities
• Outperform prior work on the precision, efficiency, and scalability of exploit generation.
THANK YOU FOR YOUR ATTENTION QUESTIONS?

NAVEX is available at
https://github.com/aalhuz/navex
REFERENCES


