Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# Refatoração e Recomposição da Integridade Estrutural: O Caso do Linderhof

Amanda Lopes Dantas

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. João José Costa Gondim

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# Refatoração e Recomposição da Integridade Estrutural: O Caso do Linderhof

Amanda Lopes Dantas

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. João José Costa Gondim (Orientador)
CIC/UnB

Prof. Dr. André Drummond
CIC/UnB

Prof. Dr. Guilherme N. Ramos
CIC/UnB

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 13 de dezembro de 2019

# Dedicatória

Eu dedico esse trabalho a minha família pelo constante apoio. Dedico, também, a todos professores, amigos e colegas pela paciência e disponibilidade.

# Agradecimentos

Agradeço ao Professor Dr. Gondim pela orientação. Agradeço ao Alan Tamer Vasques pela ajuda na gerência do projeto e pelos esforços de revisão. Agradeço ao Matheus de Oliveira Viera pela ajuda durante toda minha graduação e pelos esforços de implementação das novas funcionalidades. Agradeço à Beatriz Klink pela ajuda na revisão do trabalho. Agradeço à Letícia Lopes Dantas pela ajuda na revisão do trabalho e pelo ajuda durante toda a minha graduação. Agradeço ao Fabio Henrique Cavalcante Dantas e à Claudia Maria Lopes Dantas pelo constante incentivo e carinho.

# Resumo

A ferramenta Linderhof foi construída para implementação de ataques de negação de serviço amplificados. Suas funcionalidades foram implementas por diferentes programadores e devem ser estendidas em trabalhos futuros de graduação. A arquitetura da ferramenta se mostrou muito complexa. A implementação não segue a arquitetura proposta e o código da ferramenta pode ser melhorado. A documentação do software é escassa. Nesse trabalho, a arquitetura e a implementação da ferramenta foram reestruturadas, o código refatorado e a documentação ampliada.

**Palavras-chave:** refatoração, reestruturação, documentação, ataques de negação de serviço

# Abstract

Linderhof was created as tool for the execution of amplified denial of service attacks. It was implemented by different programmers and its functionalities will be extended in the future. The architecture of the tool is too complex and the source code does not strictly follows it. There isn't much software documentation and the source code can be improved. In this dissertation, the architecture and source code of the tool were restructured, the code refactored and more documentation elaborated.

**Keywords:** code refactoring, code restructuring, software documentation, denial of service attacks

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Lista de Abreviaturas e Siglas

**CoAP** *Constrained Application Protocol.*

**DDoS** *Distributed Denial of Service.*

**DNS** *Domain Name System.*

**DoS** *Denial of Service.*

**IDEs** *integrated development environments .*

**NTP** *Network Time Protocol.*

**RTFM** *Read the F***ing Manual.*

**SNMP** *Simple Network Management Protocol.*

**SSDP** *Simple Service Discovery Protocol.*

# Capítulo 1

# Introdução

Uma propriedade intrínseca aos softwares atuais é a necessidade de evoluir. Porém, quanto mais um software é modificado, melhorado e adaptado a novos requisitos, mais complexo ele se torna e mais a implementação se distancia da arquitetura original. Assim, boa parte do custo de desenvolvimento de software diz respeito aos esforços de manutenção. As técnicas de reestruturação e refatoração fazem parte desses esforços. A documentação, por sua vez, está relacionada com o custo dessa manutenção, sendo que, para sistemas com documentação de melhor qualidade, o custo de manutenção tende a ser menor.

## 1.1 Motivação

Em 2015, foi desenvolvida a ferramenta Striker pelo então aluno de graduação Tiago Fonseca Medeiros para o estudo de ataques *Denial of Service* (DoS) utilizando o protocolo *Simple Network Management Protocol* (SNMP) [5]. A ferramenta então teve sua segunda versão no trabalho de graduação de Henrique Senoo Hirata, com a inclusão do protocolo *Simple Service Discovery Protocol* (SSDP) [6].

Pela importância do estudo de ataques DoS e pelo crescente interesse na área de estudo por parte dos alunos de gradução da UnB, viu-se a necessidade de uma ferramenta mais abrangente. Foi então desenvolvida a ferramenta Linderhof, pelo aluno Igor Fernandes Miranda [7]. Foi criada para a ferramenta uma arquitetura modular, diferindo da arquitetura do Striker, e exportando, porém, muitos dos conceitos da sua implementação. As ferramentas diferem, também, em linguagem de programação, sendo o Striker implementado em Java e o Linderhof em C.

Inicialmente, o Linderhof foi implementado para estudar a dinâmica de ataques de negação de serviço por reflexão amplificada, explorando o Memcached. Porém, o Linderhof foi logo expandido para também explorar os protocolos *Network Time Protocol*

(NTP)[8], *Domain Name System* (DNS), *Constrained Application Protocol* (CoAP)[9] e *Simple Service Discovery Protocol* (SSDP). O suporte para cada um desses protocolos foi implementado por alunos diferentes, em seus respectivos trabalhos de graduação, cada um em um *fork* da implementação original, sendo o SSDP exportado da segunda versão do Striker. Ao todo, a ferramenta faz parte de outros 6 trabalhos de graduação e uma tese de mestrado, sendo que alguns desses ainda não foram defendidos.

## 1.2 Justificativa

A ferramenta Linderhof foi construída com o intuito de ser colaborativa, porém, ao longo do tempo de trabalho, várias funcionalidades foram acrescidas e retiradas. O código original foge, em alguns aspectos, da arquitetura projetada para a ferramenta. Não existia uma documentação de código e a pouca documentação das funcionalidades era escassa e confusa.

Ao todo, 6 programadores trabalharam na ferramenta até agora, cada um em seu *fork* específico, porém existe um planejamento para trabalhos futuros envolvendo mais programadores. Os trabalhos de graduação que fizeram uso do Linderhof foram todos apresentados em um período de um ano e meio. Assim, a comunicação entre os programados que fizeram modificações na ferramenta foi facilitada até agora, porém isso deve se modificar dentro de alguns semestres.

## 1.3 Objetivo

O objetivo do trabalho é refatorar e reestruturar o código do Linderhof e documentar a ferramenta. Assim, a ferramenta Linderhof deve passar por uma reestruturação de código, simplificando sua arquitetura. O código da ferramenta será documentado e a documentação de funcionalidades já existente será expandida. Durante o processo de documentação, a ferramenta passará por uma refatoração de código.

Esta ferramenta foi, e está sendo desenvolvida, com fins didáticos e acadêmicos, para demonstração de vulnerabilidades e o desenvolvimento de melhorias na mitigação de ataques.

## 1.4 Organização do Texto

Esse trabalho está organizado da seguinte forma:

- O capítulo 2 aborda o fundamento teórico necessário para o entendimento do trabalho;

- O capítulo 3 apresenta a ferramenta original;
- O capítulo 4 apresenta o processo de refatoração, reestruturação e documentação da ferramenta;
- O capítulo 5 apresenta a conclusão e as propostas para trabalhos futuros.

# Capítulo 2

# Referencial Teórico

## 2.1 Ataques *Denial of Service* (DoS)

Um ataque DoS é um ataque com o objetivo de impedir o acesso de usuários legítimos aos serviços fornecidos pela vítima. [10]. Existem vários contextos onde um ataque DoS pode ocorrer, como em sistemas operacionais e serviços baseados em rede.

A classificação de ataques DoS mais completa e comumente usada é a taxonomia apresentada por Specht et al. [11]. Nela, são definidos dois tipos de ataques DoS: Esgotamento de Largura de Banda e Esgotamento de Recursos. Um ataque *Denial of Service* (DoS), do tipo esgotamento de largura de banda, consiste em saturar a vítima com tráfego ilegítimo, para que o tráfego legitimo não consiga alcançá-la. Nesse caso, qualquer sistema que depender dos links de comunicação congestionados no caminho do ataque sofrerá danos, ou seja, não só a vítima será atacada [12].

Um ataque *Denial of Service* (DoS) de esgotamento de recursos compromete os recursos do sistema da vítima, o que a impede de processar as requisições de usuários legítimos. Isso porque, quando a vítima é congestionada, ela rejeita pacotes e informa aos remetentes, que reduzam sua taxa de envio. Assim, usuários legítimos diminuirão sua taxa de envio, enquanto o atacante aumenta, ou mantém, sua própria taxa. Eventualmente, os recursos da vítima, como CPU e memória, não irão conseguir atender às requisições legítimas. Geralmente, a vítima é um *Web server* ou *proxy* conectado à *Internet.*[12]. A Figura 2.1 ilustra essa classificação.

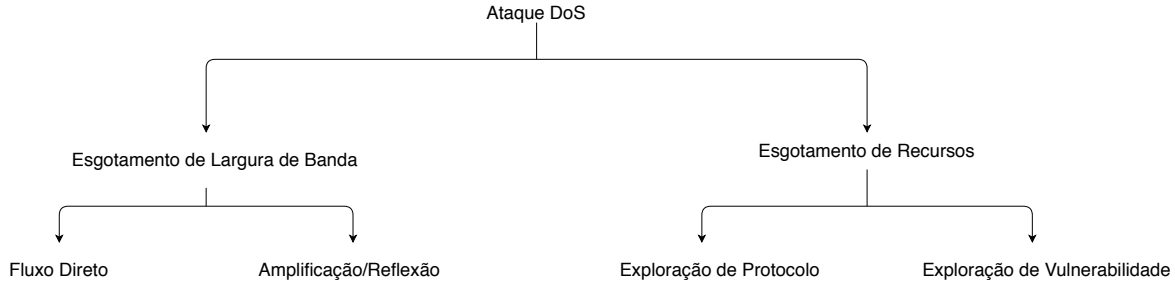


Figura 2.1: Taxonomia de ataques DoS

Comumente, porém, uma forma mais direta de classificação é usada fora do meio acadêmico. Um exemplo disso é a classificação da Cisco [13], que divide os ataques em 3 principais tipos: ataques baseados em volume, ataques de aplicação (usam a camada de aplicação para atacar) e ataques de exploração de protocolos. A Figura 2.2 ilustra essa classificação.

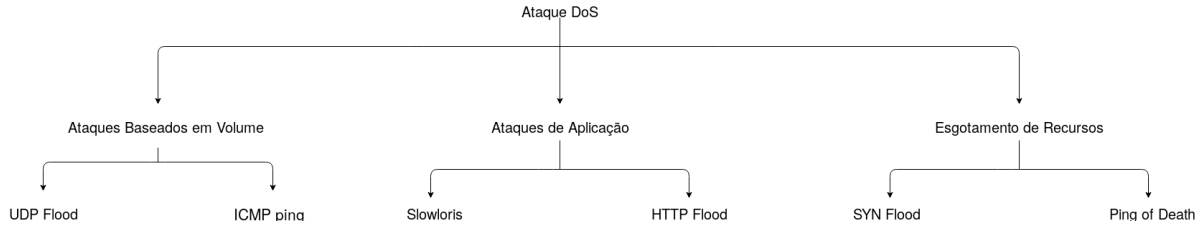


Figura 2.2: Taxonomia de ataques da Cisco

### 2.1.1 Ataques *Distributed Denial of Service* (DDoS)

Um ataque *Distributed Denial of Service* (DDoS) ocorre quando vários dispositivos coordenados atacam uma ou mais vítimas, com o intuito de exaurir os recursos de processamento ou conectividade da vítima.

Existem três entidades envolvidas na estrutura do ataque: o atacante, os zombies e a vítima. *Zombies* são máquinas comprometidas que obedecem ao atacante e direcionam o ataque à vítima. A estrutura básica de um ataque DDoS é exemplificada na Figura 2.3, porém os diferentes tipos de ataque podem modificar essa estrutura.

### 2.1.2 *Ataques por Reflexão/Amplificação*

Um ataque DoS por reflexão tem o objetivo de mascarar a fonte do ataque, usando terceiros para repassar tráfego ilegítimo para a vítima. Esses terceiros são denominados refletores [12]. A Figura 2.4 exemplifica como a reflexão modifica a estrutura básica de um ataque DDoS.

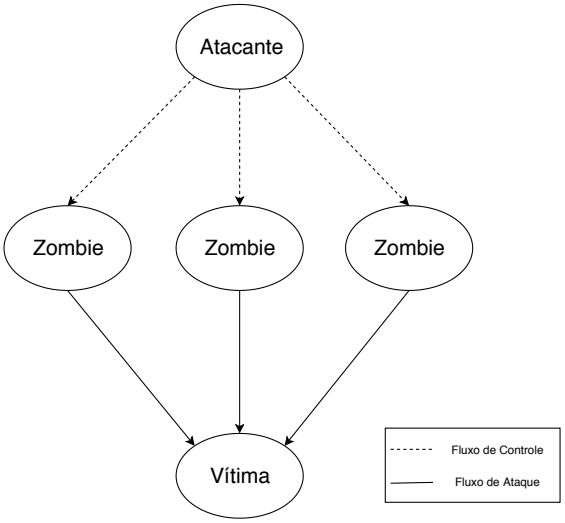


Figura 2.3: Estrutura básica de um ataque DDoS

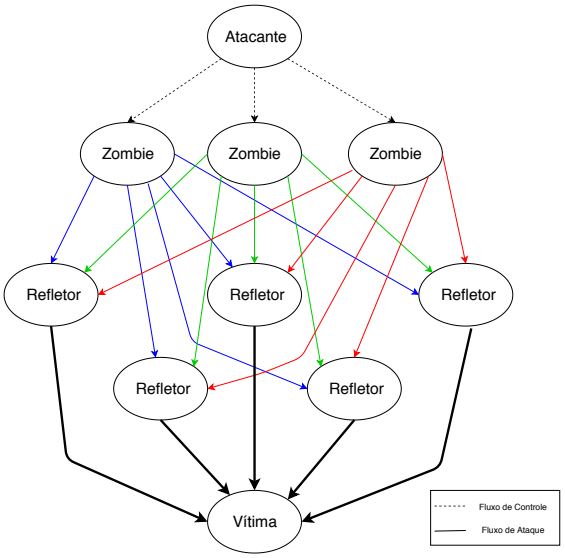


Figura 2.4: Estrutura de um ataque DDoS com Reflexão

O atacante manda para o refletor pacotes com o endereço de origem igual ao IP da vítima. Assim, quando o refletor responder aos pacotes, ele direciona o tráfego à vítima. Nesse ataque, geralmente são usados protocolos onde o pacote de *reply* é maior que o pacote de *request*, assim, o ataque também gera um efeito de amplificação. O uso de refletores dificulta o rastreamento da origem do ataque. A Tabela 2.1 ilustra os protocolos mais comuns em ataques de reflexão/amplificação e suas estatísticas, obtidas pela Arbor Networks, entre a segunda metade do ano de 2018 [3].

Tabela 2.1: Estatísticas de Ataque por Reflexão [3]

| **Protocolo** | **Total de Ataques** | **Volume do Maior Ataque (Gbps** |
|---|---|---|
| DNS | 251,355 | 388 |
| NTP | 206,586 | 260 |
| SSDP | 79,960 | 287 |
| Chargen | 22,150 | 128 |
| TCP SYN/ACK | 21,628 | 156 |
| SNMP | 18,523 | 210 |
| rpcbind | 9,011 | 121 |
| memcached | 5,125 | 245 |
| mDNS | 1,616 | 186 |
| MS SQL | 1,593 | 75.8 |
| NetBIOS | 856 | 121 |
| RIPv1 | 293 | 64.7 |

### 2.1.3 Mitigação de Ataques *Denial of Service* (DoS)

No momento em que os danos de um ataque DoS passam a ser sentidos, geralmente não há o que se fazer, senão desconectar a vítima da rede e, manualmente, reparar o problema. Assim, existem muitas técnicas propostas para mitigação e detecção desses ataques. De preferência, o ataque deve ser bloqueado o mais perto possível da sua origem [14]. Já foram propostas diversas técnicas de mitigação de ataques DoS e, como de praxe, ao longo dos anos, os ataques foram evoluindo para evadir tais técnicas.

Existem três grandes categorias de defesa contra ataques DoS: prevenção de ataque, detecção de ataque, identificação da fonte e reação ao ataque. Porém, um mecanismo de mitigação robusto deve empregar todas as três categorias.

Técnicas de prevenção estão espalhadas ao longo de toda rede. Em sua grande maioria, essas técnicas buscam solucionar vulnerabilidades que podem ser usadas para lançar um ataque DoS. Alguns exemplos de técnicas de prevenção são: proteção *fail-safe*; realocação de recursos; filtros de tráfego [15, 16] e protocolos para identificação de usuários legítimos [17].

O maior desafio para as técnicas de detecção de ataques são os falsos positivos. Isso se deve ao fato do tráfego de um ataque DoS se parecer muito com tráfego legítimo ( as chamadas *flash crowds*). As técnicas de detecção ajudam a vítima a ganhar mais tempo para proteger os seus usuários legítimos e responder ao ataque. Às vezes, essas técnicas se baseiam em características do ataque para detectá-lo. Outras vezes, podem se basear na modelagem do comportamento normal da rede e procurar por anomalias [18, 19].

Idealmente, após a detecção do ataque, bastaria bloquear o tráfego com a mesma origem, mitigando-o. Porém, não existe nenhuma maneira fácil de detectar a origem do ataque. Pois, a informação do endereço da fonte pode ser forjada com facilidade, e, por natureza, IP é um protocolo sem estado. As técnicas de identificação da porta de origem são, em geral, algoritmos *traceback* [20]. Alguns estudos também sugerem novos protocolos, ou a modificação de protocolos vigentes, para que informações de estado sejam guardadas ao longo da rede [21].

Por fim, esquemas de reação ao ataque devem ser desenvolvidos para minimizar os danos enquanto o ataque está acontecendo. Exemplos de resposta são: a implementação de gargalo intencional no sistema, que limita a taxa de injeção na vítima [22], e a filtragem de pacotes *upstream*, com o endereço de origem identificado. Outra forma de reação é o modelo em que os servidores encorajam os clientes de tráfego legítimo a aumentar suas taxas, para sufocar o tráfego ilegítimo [23].

## 2.2 Reestruturação e Refatoração

### 2.2.1 Refatoração

A refatoração é o processo de mudar um sistema de software, de maneira que o comportamento externo do código não seja afetado, mas sua estrutura interna melhore. É uma forma disciplinada de limpar um código, minimizando as chances de introduzir bugs [4]. O objetivo principal da refatoração é melhorar a legibilidade do código, aumentando sua qualidade. Como existem diversos tipos de refatoração, o efeito na qualidade pode ser estimado, classificando as mudanças feitas nos termos dos atributos de qualidade que elas afetaram. Ou seja, algumas refatorações removem redundâncias, outras melhoram a reusabilidade, enquanto outras aumentam o nível de abstração, porém todas melhoram a qualidade do código.

A definição de refatoração é muito próxima da definição de reestruturação. Porém, refatoração diz respeito necessariamente ao código e, em geral, ao paradigma de orientação a objetos[24]. Ou seja, a ideia principal é redistribuir classes, variáveis e métodos, para facilitar adaptações futuras e extensões.

Um grande esforço da refatoração é a limpeza de código morto. Código morto é todo aquele código presente no arquivo, mas que nunca é executado. Geralmente, código morto consiste-se em código legado, que foi substituído, mas não apagado, apenas comentado. A presença de código morto diminui a legibilidade e pode causar confusão.

Uma das técnicas mais comuns de refatoração é a detecção de *code smells*. Os *code smells* são uma metáfora criada por Kent Beck [4]. Nela, os chamados *bad smells* são os

padrões de implementação geralmente relacionados a design ruim, ou a maus costumes de programação. Os pontos de *bad smells* são pontos que, potencialmente, devem ser refatorados. A Tabela 2.2 apresenta os *bad smells* mais comuns e a refatoração recomendada por Beck [4].

Tabela 2.2: Bad smells [4]

| **Bad Smell** | **Descrição** | **Refatoração** |
|---|---|---|
| Mysterious Name | Classe, método ou atributo com nome não intuitivos | Renomear a classe, método ou atributo em questão |
| Código Duplicado | Mesma estrutura de código em mais de um lugar | Criar uma função; Mudar o código de lugar |
| Funções Longas | Geralmente, funções que fazem mais do que seu nome indica | Extrair outra função; Procurar por código dentro da função que possa ser refatorado |
| Muitos parâmetros | Funções com longas listas de parâmetros | Substituir parâmetros por querys; remover argumentos de flag; combinar parâmetros em estrutura de dados |
| Divergent Change | Um módulo muda com frequência por motivos diferentes | Dividir o módulo |
| Shotgun Surgery | Uma alteração pede pequenas mudanças em outras classes | Mover funções e campos para criar um módulo; Combinar funções em classes |
| Feature Envy | Um módulo se comunica mais com funções e dados de outro módulo | Mover funções entre os módulos |
| Lazy Element | Estruturas desnecessárias | Mudar a hierarquia; extinguir a estrutura |
| God Class | Classe responsável por muito no sistema | Dividir a classe em duas; Criar uma subclasse |
| God Method | Centraliza toda a funcionalidade da classe | Dividir o método em dois; Transformar o método em classe |
| Refused Bequest (Legado Rejeitado) | Uma classe herda atributos e métodos de outra classe, mas não os usa | Mover o método ou atributo da superclasse para a subclasse; Substituir o relacionamento de herança por associação |

É importante ressaltar que a refatoração de um código deve ser feita em incrementos pequenos. Pois, uma grande refatoração tem maior chance de gerar *bugs*. Idealmente, os incrementos da refatoração devem ser pequenos o suficiente para ser facilmente reversíveis, caso necessário. A implementação de testes em todo o código envolvido na refatoração também é necessária para garantir que a execução do mesmo continue inalterada após as mudanças.

### 2.2.2 Reestruturação

A reestruturação é a transformação de uma forma de representação em outra, no mesmo nível relativo de abstração, preservando o comportamento externo do sistema [24]. Ou seja, é uma transformação de aparência. Ela cria novas versões do sistema, mas não envolve novos requisitos nessas modificações. Porém, pode levar a observações que sugerem mudanças para melhoria de alguns aspectos do sistema. Na reengenharia de software, reestruturação é comumente usada para converter código legado em um código mais modular ou estruturado, para migrar de linguagem de programação, ou trocar de paradigma.

A definição de comportamento é complexa. Muito comumente, a ideia de preservar o comportamento de um software, presente na definição de reestruturação, pode ser simplificada em "para o mesmo *input*, deve-se obter o mesmo *output*". Porém, em alguns domínios de aplicação, essa simplificação não representa o total do que deve ser preservado. Para sistemas em tempo real, por exemplo, um aspecto importante do seu comportamento é a execução ordenada de certas sequências de operações. Já, para sistemas de segurança críticos, é importante que seu comportamento preserve a sua noção de segurança.

Além disso, o comportamento que deve ser preservado nem sempre é o mesmo para todas as estruturas do código. Assim, tudo que for identificado como parte do comportamento que deve ser preservado pela reestruturação (ou refatoração) deve fazer parte de um conjunto de testes. Se o comportamento for o esperado para esse conjunto de testes, após feita a reestruturação (ou refatoração), é uma evidência que o comportamento definido foi preservado.

Existem diversas técnicas para auxiliar na reestruturação, a Visualização de Software é uma delas. Essa técnica propõe usar diagramas e outras ferramentas gráficas para identificar pontos de reestruturação. A técnica de *Meta Modelling* também pode ser usada para identificar tais pontos. Uma das vantagens de usar essa técnica é que ela torna a refatoração menos dependente da linguagem de programação usada na implementação. Basicamente, o objetivo geral de um *meta modelling* é criar um modelo que governa como outros modelos devem ser compostos. A modelagem de software em grafos, com a reestruturação representando transformações, também é muito comum.

A reestruturação pode ser aplicada para diversos níveis de abstração, como na arquitetura do software e nos modelos de design. No nível de arquitetura, alguns autores recomendam uma refatoração baseada diretamente na representação gráfica [25], enquanto outros recomendam que a reestruturação da arquitetura seja feita por meio de refatorações menores [26].

Atualmente, existem muitas ferramentas disponíveis para automatizar vários aspectos da manutenção de software. A reestruturação, assim como a refatoração, é um desses aspectos.

## 2.3 Documentação

Documentação pode ser definida como qualquer artefato cujo propósito é comunicar informações sobre o sistema de software ao qual pertence [27]. Existem diversos tipos de documentação, que variam de acordo com o projeto. Alguns tipos de documentação são: requisito, especificação, arquitetura e detalhamento de design. O objetivo principal de qualquer documentação, porém, é bem simples: comunicação. Toda documentação é escrita com o intuito de partilhar alguma informação com alguém.

Nos campos mais tradicionais da engenharia, a documentação é parte fundamental em todos os momentos de desenvolvimento de um projeto. Essa documentação pode ser vista em artefatos como plantas de casas, projetos de instalações elétricas, diagramas de circuitos e detalhamento de projetos mecânicos, por exemplo. O mesmo não é verdade, porém, para a engenharia de software. Muitas vezes, a maioria da documentação é produzida ao final do desenvolvimento e negligenciada antes, e durante, o processo de implementação do software.

Assim, na prática, a documentação de software costuma ser escassa e incompleta, apesar de a maioria das metodologias de desenvolvimento exigirem a escrita de documentação. O custo da produção de documentação é facilmente medido, porém o custo da falta de documentação não é. Isso porque ele está espalhado em vários outros custos, como os custos das mudanças, de achar bugs, de falhas de segurança, atrasos de produção, entre outros.

Documentação ruim é causa de muitos erros e reduz a eficiência em todas as fases de uso e desenvolvimento. Já uma boa documentação, pode melhorar o reuso de designs antigos, tornar mais fácil a integração de módulos escritos separadamente, facilitar as inspeções de código, melhorar o processo de testes e aumentar a eficiência das correções e melhorias.

A documentação é uma parte controversa do desenvolvimento. Uma documentação importante para o arquiteto de um sistema pode não ser tão importante para o desenvol-

vedor, para o cliente, ou para o usuário final. Muito provavelmente, cada um dos agentes envolvidos no ciclo de vida do software terão necessidades diferentes de informações, bem como de documentação. O crescimento do movimento *open-source* aumentou a cultura de documentação, além de mudar o foco de sua metodologia.

Um efeito colateral da documentação é a melhora da qualidade geral do software. Nesse sentido, uma documentação detalhada exige um exame detalhado do código. Existem diversos métodos de escrever documentação e, em conjunto com a metodologia de desenvolvimento do projeto, são o que ajuda a definir quais documentos devem ser escritos e para quem. Algumas regras gerais para a documentação de código são [28]:

1. Escrever da perspectiva do leitor
2. Evitar repetições desnecessárias
3. Evitar ambiguidade
4. Usar uma organização padrão
5. Registrar as tomadas de decisão
6. Manter a documentação suficientemente atualizada
7. Revisar a documentação adequando-a ao fim a que ela se destina

### 2.3.1 Classificação de Documentos

Segundo Parnas [29], existem dois tipos principais de documentação: narrativa tutorial e obras de referência. Narrativas tutoriais são, geralmente, projetadas para ser lidas por completo. São direcionadas ao usuários com pouco conhecimento anterior sobre o projeto. Já, obras de referência têm o objetivo de ajudar o leitor a recuperar fatos específicos do documento. São direcionadas a usuários que já conhecem o projeto e procuram responder dúvidas, ou cobrir lacunas no seu conhecimento.

Todas as obras de referência são descrições, sendo que uma descrição é um artefato que apresenta propriedades de um produto que existe. Um tipo especial de descrição são as especificações, que apresentam as propriedades necessárias de um produto que ainda não existe. Geralmente, documentos escritos a partir da inspeção de um produto são descrições. Esse documentos só podem expressar o que foi feito, não o que se desejava fazer, ou o que foi requisitado. Ou seja, as especificações devem ser construídas antes, e durante, a implementação.

### 2.3.2 Metodologias de Documentação

#### Visão Clássica

Um dos métodos de escrever documentação é o método por *views* [28]. Cada *view* é considerada uma parte do sistema podendo ser vista como módulo, por exemplo. Essa metodologia produz uma documentação completa com foco na arquitetura do sistema e é destinada a sistemas maiores. São recomendados dois conjuntos de documentos: os que compõem a documentação de cada view e os que compõe a documentação do sistema. A descrição dos documentos necessários para o métodos de views pode ser encontrada na Tabela 2.3 [28].

Tabela 2.3: Documentos Sugeridos para o Método por *Views*

| | |
|---|---|
| Documentação de *View* | Representação, geralmente gráfica, que descreve os elementos principais e seu relacionamento |
| | Catálogo de elementos: que explica e define os elementos presentes e uma lista de suas propriedades |
| | Especificação das interfaces dos elementos e seu comportamento |
| | Guia de variabilidade explicando qualquer mecanismo presente no sistema que possa adaptar a arquitetura |
| | Informações de tomada de decisão e de design |
| Documentação do Sistema | Introdução para todo o pacote, incluindo um sumário que ajuda um *stakeholder* a localizar a informação desejada rápido |
| | Descrição do relacionamento entre os módulos e os relacionamentos do sistema como um todo |
| | Restrições e informações de tomada de decisão sobre toda a arquitetura |
| | Qualquer informação de gestão que seja importante para manutenção do pacote |

Parnas [29] apresenta uma metodologia mais geral de documentação, seguindo os padrões comuns na engenharia de software. Ele defende que não sejam usadas ferramentas de automatização de documentação e, por sua vez, lista documentos sempre necessários em qualquer sistema:

- Documento de requisitos do sistema: que é uma representação caixa-preta do sistema, identificando todos os *inputs* e a sua relação entre os valores de output.
- Documento de estrutura de módulo: que deve conter a hierarquia da decomposição do sistema em módulos e a responsabilidade de cada módulo.
- Documentos de interface de módulo: que devem conter estruturas de dados, efeito nos dados, abstração de funções e relacionamentos.
- Documentos de função do programa: que descreve os efeitos de executar o programa em elementos externos de dados como, por exemplo, um mapa de todos os estados até o estado final, ou um mapeamento de eventos e seus outputs.

A maioria dos métodos de documentação clássica fazem uso de diagramas e modelos padronizados, utilizando linguagens como UML, entre outros. É comum equipes adotarem regras rígidas para documentação que definem quais documentos devem ser escritos e quando, no processo de desenvolvimento, devem ser elaborados. Porém, a obrigatoriedade e a rigidez no processo de documentação são, em grande parte, responsáveis pela baixa qualidade da documentação escrita.

### Código Como Documentação

A filosofia de *"code as documentation"* acredita que o código fonte deve ser a fonte primária da documentação. Essa filosofia é muito comum em métodos ágeis de desenvolvimento e em *eXtreme Programming* . O princípio básico desse método é que o código fonte é o único artefato detalhado e preciso o suficiente para cobrir todas as necessidades de documentação.

Reeves [30] defende que o objetivo final de qualquer projeto de design em engenharia é o de produzir alguma forma de documentação. Assim, defendendo a ideia do código fonte como design, a documentação principal seria o próprio código fonte, que deve ser limpo e fácil de ler, e os comentários nele, que devem, por sua vez, descrever as ideias que fazem parte daquele elemento de design.

A única documentação que deve ser produzida é a documentação auxiliar. Essa deve capturar informações importantes para o espaço de problema, que não são encontradas diretamente no design, assim como os aspectos do design que são difíceis de extrair diretamente. Inclui, portanto, todos os aspectos melhor descritos graficamente, já que é difícil incluí-los em comentário no código fonte. Também, são necessárias as documentações que não fazem parte do design de um código, como manuais do usuário e guias de instalação. Reeves [30] defende que idealmente deve ser usada automação para gerar essa documentação. É recomendado que a documentação auxiliar seja mínima e informal.

### Documentação *Open-Source*

Entre os princípios básicos para o software livre, a produção colaborativa e a transparência dizem respeito à documentação [31]. A documentação é necessária para produção colaborativa, pois registrar e disponibilizar mudanças, decisões e objetivos é a garantia que o trabalho individual irá influenciar positivamente o conjunto. A transparência pede que a comunidade tenha acesso ao projeto e possa conhecer todos os seus aspectos. A documentação é, assim, uma das maneiras de tornar o projeto acessível, sendo, também, muito usada para divulgação dos projetos de software livre.

É muito comum que desenvolvedores novos ao movimento *open-source* comecem a contribuir por mudanças na documentação, já que mudá-la é bem mais simples do que mudar um código. Assim, muitos projetos investem em documentação para atrair esses contribuintes de primeira viagem. Alguns projetos até encorajam que todos os contribuintes novos comecem com *commits* na documentação, já que é uma ótima forma de conhecer o código.

Grande parte da documentação *open-source* é escrita para o usuário final e para educar os futuros contribuintes do projeto que, muitas vezes, são a mesma pessoa. Como os projetos variam em tamanho, também existem várias divergências sobre que documentação deve ser escrita. No geral, um projeto *open-source* deve ter pelo menos os arquivos [32]:

- README: que deve conter a explicação do projeto, as dependências, as instruções de uso, o processo de instalação e a licença [33]
- CONTRIBUTING: onde devem estar instruções para potenciais contribuintes.
- CODE_OF_CONDUCT: que reflete os valores da comunidade do projeto e descreve as expectativas para os contribuintes e mantenedores.

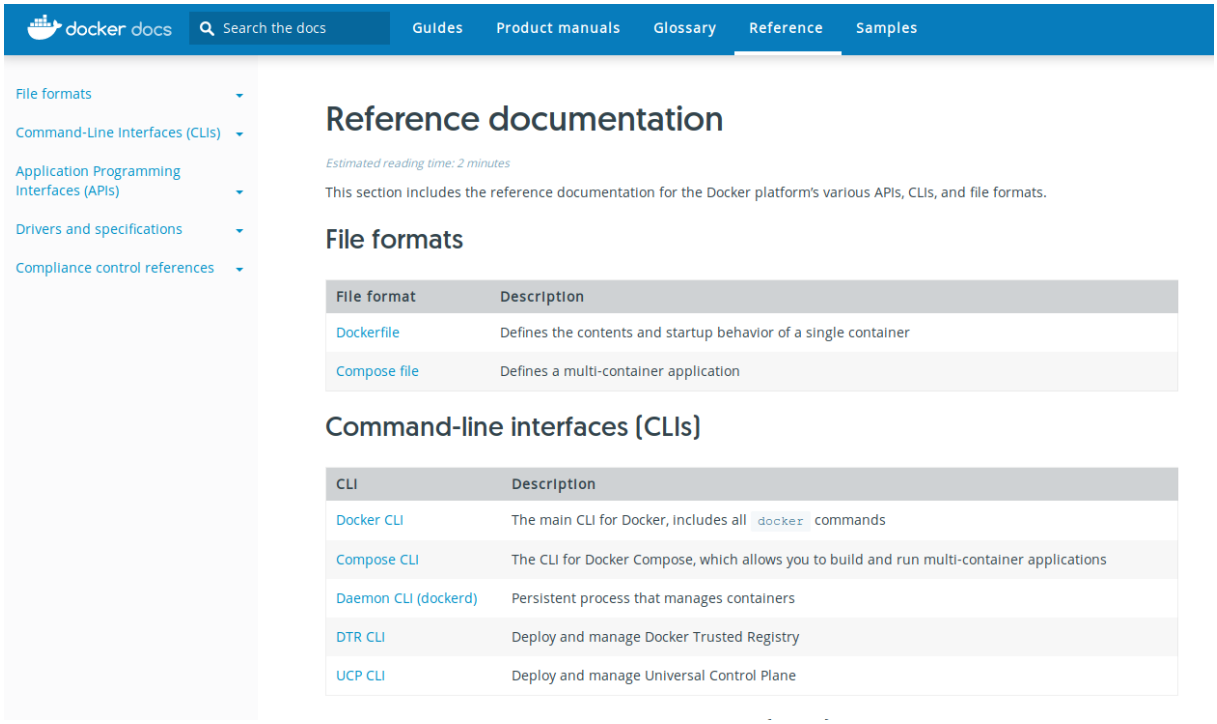


Figura 2.5: Exemplo de *reference file* [1]

Muito provavelmente, projetos de sucesso irão copiosamente expandir essa documentação. São comuns em projetos *open-source* documentos explicando o código e sua implementação, chamados de *reference files*. Documentações em formato wiki são bastante comuns, e ferramentas de hospedagem e controle de software já disponibilizam as wikis como *feature*. A documentação em *open-source* também é muito usada para definir os próximos passos do projeto, mostrando o que falta ser desenvolvido para cumprir uma meta ou objetivo, listando os bugs e melhorias ainda não abordados [34] .

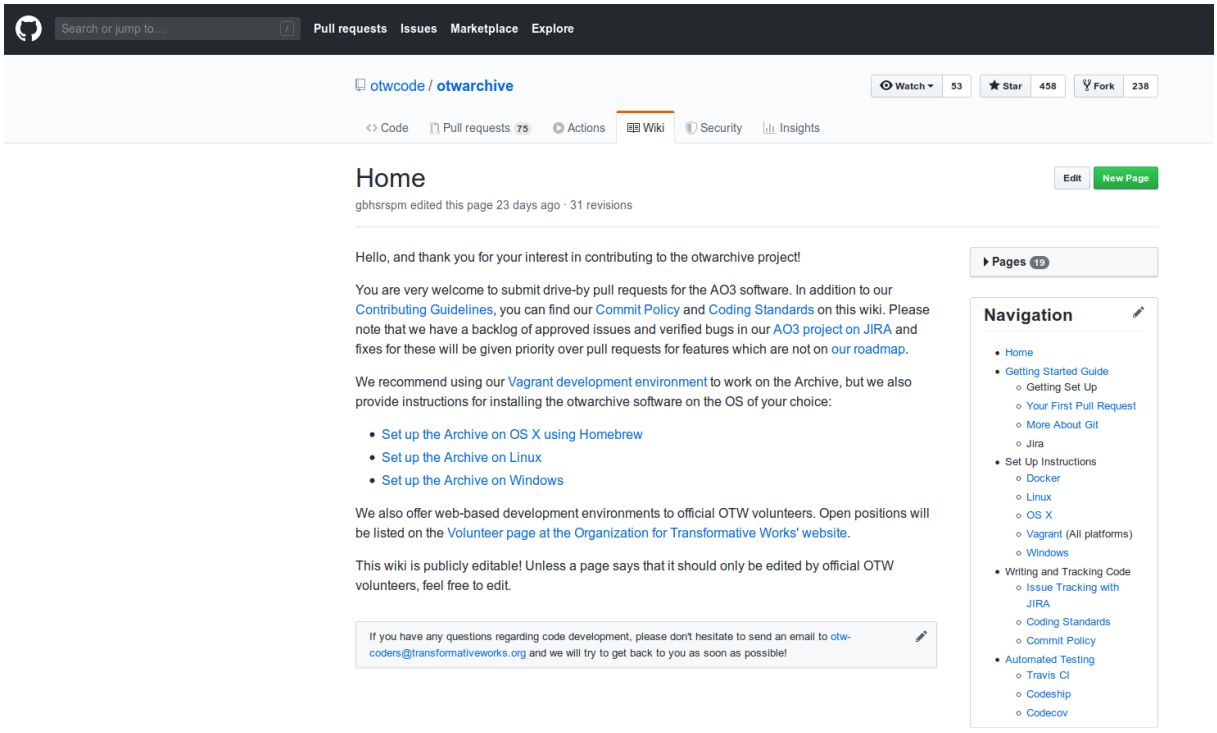


Figura 2.6: Estrutura de Documentação em wiki [2]

Berglund e Priestley [35] defendem que o processo de documentação em projetos de software livre é contínuo, já que conteúdo é constantemente criado em listas de email e FAQs que, por sua vez, representam debates técnicos nas comunidades de usuários e respondem questões sobre os produtos, ajudando a direcionar o desenvolvimento futuro.

A documentação em um projeto *open-source* deve responder às perguntas do usuário e não, necessariamente, descrever todo o sistema. Berglund e Priestley [35] defendem que a falta de descrição na documentação de um projeto pode ser vista como fonte de conhecimento. Se os usuários não requisitaram documentação para uma determinada *feature*, o design da interface deve ser óbvio, ou a *feature* não deve estar sendo usada. Nos dois casos, a documentação é desnecessária.

O processo de documentação *open-source* disponibiliza um jeito de medir áreas de uso e tipos de interação sendo, assim, valioso para o processo de desenvolvimento. Como os usuários participam do processo de documentação, com conteúdo de discussões, perguntas e respostas, eles também estão contribuindo para o processo de levantamento de requisitos futuros.

A documentação de software pode ser vista como intrínseca à cultura *open-source*, já que o usuário é estimulado a obter conhecimento para resolver e buscar soluções sozinho. A comunidade reenforça a necessidade de uma documentação em suas interações com outros usuários, sendo as respostas *Read the F***ing Manual* (RTFM) e *"google it"* muito

comuns para perguntas básicas em fóruns e chats. A própria expressão RTFM pressupõem a existência de informações suficientes e disponíveis a usuários leigos e avançados.

# Capítulo 3

# Linderhof

O objetivo da ferramenta Linderhof é suportar o *benchmarking* de soluções de mitigação de ataques DDoS refletidos. A ferramenta é dual e tem sido usada no estudo da dinâmica dos ataques DDoS por reflexão amplificada. Seu nome vem do Palácio de Linderhof, na Alemanha, conhecido pelo seu grande corredor de espelhos (*hall of mirrors*). A ferramenta faz alusão justamente a esse *hall of mirros*. Em metáfora, cada protocolo implementado na ferramenta é chamado de *mirror*. Na sua versão original, a ferramneta é dividida em três módulos base de acordo com a Figura 3.1. A arquitetura foi implementada na pasta do src do projeto que é organizada de acordo com a Tabela 3.1. Cada um desses módulos é apresentado abaixo.

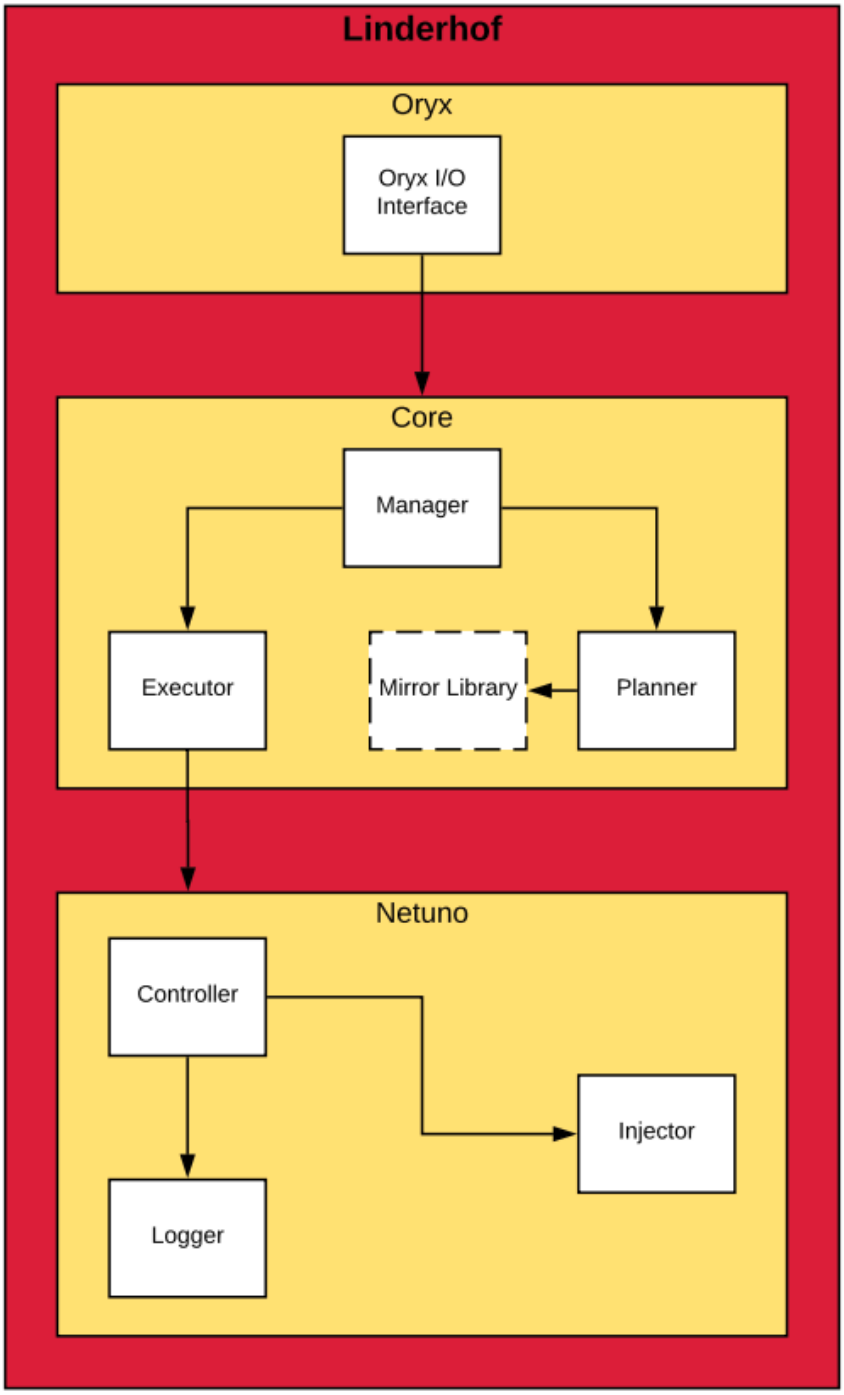


Figura 3.1: Arquitetura do Linderhof

Tabela 3.1: Organização da pasta src antes da refatoração.

| Caminho | Arquivos | Descrição |
|---|---|---|
| common | blacksmith.c<br>capabilityHelper.c<br>cliparser.c<br>error.c<br>list.c<br>memutils.c<br>netio.c<br>pthreadHelper.c<br>signalsHandler.c<br>timeHelper.c | Arquivos usados em mais de um módulo |
| include/commander<br>include/common<br>include/hom<br>include/interface<br>include/monitor<br>include/netuno | *.h | Headers |
| include | venus.h | Implementação da estrutura draft |
| interface | interface.c<br>oryx.c<br>snowman.c | Oryx |
| linderhof/commander | executor.c<br>linderhof.c<br>planner.c | Core |
| linderhof | main.c | N/A |
| monitor | crake.c<br>main.c | N/A |
| netuno | injector.c<br>logger.c<br>netuno.c | Netuno |
| test | *.h<br>*.c | N/A |

A cor vermelha representa código morto.

## 3.1 Módulo Oryx

O módulo Oryx é responsável pela interface com o usuário. O Linderhof aceita vários parâmetros por linha de comando representados na Figura 3.2. Esses parâmetros são traduzidos para um plano de ataque, chamado *draft* na implementação, que é encaminhado para o Módulo Commander. A estrutura do draft contém informações como o tipo de mirror, o ip do refletor, o timer do ataque, nível do ataque e as portas do refletor e do amplificador.

| Nome | Opção longa | Opção curta | Opção obrigatória | Argumento obrigatório | Valor padrão | Descrição |
|---|---|---|---|---|---|---|
| ARG_MIRROR | mirror | m | Sim | Sim | Não possui | Espelho que será utilizado no ataque |
| ARG_TARGETIP | target | t | Sim | Sim | Não possui | IP do alvo |
| ARG_AMPLIFIERIP | amplifier | a | Sim | Sim | Não possui | IP do amplificador |
| ARG_TARGPORT | targport | g | Não | Sim | 80 | Porta do alvo |
| ARG_AMPPORT | amport | p | Não | Sim | Porta padrão do mirror | Porta do amplificador |
| ARG_LEVEL | level | l | Não | Sim | 1 | Level do ataque |
| ARG_TIMER | timer | c | Não | Sim | Infinito | Tempo do ataque |
| ARG_LOGFILE | log | f | Não | Sim | NULL | Nome do arquivo de log |
| ARG_INCREMENT | inc | i | Não | Sim | FALSO | Ataque incremental |

Figura 3.2: Opções da interface

O módulo é implementado em src/interface por três arquivos como representado pela Tabela 3.1. O arquivo snowman.c não está conectado ao resto do programa representando, então, código morto. Assim, a implementação foi divida em dois arquivos: interface.c e oryx.c.

## 3.2 Módulo Core

O módulo é referenciado na implementação e na documentação como Commander. Ele pode ser visto como núcleo da ferramenta, já que é responsável por criar o planejamento do ataque de acordo com o *draft* recebido e chamar o injetor. Esse módulo é dividido em três partes: Manager, Planner, Executor e Mirror Library.

A Mirror Library é chamada Hall of Mirrors na implementação e na documentação. Os protocolos explorados na ataque são chamados de mirrors e o conjunto deles é o Hall of Mirrors. Cada mirror têm sua função de chamada que é sua conexão com o resto do programa. Os mirrors são responsáveis por criar o pacote repassado para o injetor, preparar o refletor antes do ataque, e chamar o injetor.

O Manager gerencia o Planner e o Executor. Ele também inicializa a ferramenta atribuindo os *handlers* de sinal, as funções de erro e as Linux *capabilities*.

O Planner monta a estrutura do plano de ataque de acordo com o protocolo escolhido. Esse plano de ataque é uma estrutura que contém o tipo de mirror, a função de chamada do mirror e os dados necessários para executar essa função de chamada. O Executor executa o mirror fazendo uso da função de chamada alocada no plano de ataque.

O módulo é implementado em src/linderhof/commander por três arquivos como representado na Tabela 3.1. A parte manager foi implementada como linderhof.c. As partes Executor e Planner foram implementadas em arquivos de mesmo nome.

O Hall of Mirrors foi implementado em src/linderhof/hom como representado na Tabela 3.1. Ele é composto por 5 protocolos: *Constrained Application Protocol* (CoAP), *Domain Name System* (DNS), Memcached, *Network Time Protocol* (NTP) e *Simple Service Discovery Protocol* (SSDP). Cada um desses mirrors foi implementado em um *fork* diferente do projeto original, a Tabela 3.2 representa a organização de cada dessas implementações.

Tabela 3.2: Organização dos mirrors antes da refatoração.

| **Caminho** | **Arquivos** |
|---|---|
| coap/src | coapforge.c<br>coapforge.h<br>manager.c<br>strix.h |
| DNS/src | DNSforge.c<br>DNSforge.h<br>DNSmanager.c<br>DNSmanager.h |
| memcached/src | manager.c<br>memcachedforge.c<br>memcachedforge.h<br>strix.h |
| Ntp/src | NTPforge.c<br>NTPforge.h<br>NTPmanager.h<br>strix.h |
| ssdp/src | listssdp.c<br>listssdp.h<br>manager.c<br>ssdpLib.c<br>ssdpLib.h |

## 3.3 Módulo Netuno

O módulo Netuno é o injetor de pacotes da ferramenta. Assim, sua responsabilidade é a de manter a injeção de pacotes constante durante o ataque.

Cada injetor, representado por uma *thead* injetora, possui um *bucket* que corresponde à quantidade de pacotes que devem ser enviados. Esses *bucket* é implementado com uso de três variáveis: bucketMax que representa a quantidade máxima de pacotes; bucketSize que representa o tamanho atual do bucket; e freeBucket que representa o interrupção ou a continuação do ataque.

A parte Controller é responsável por controlar a taxa de injeção de cada injetor através da atualização do *bucket* dos injetores de acordo com as informações do ataque. O timer, o nível e o modo incremental são os parâmetros que interferem nessa taxa. O timer representa a duração do ataque em segundos. O nível corresponde a intensidade do ataque e segue a Tabela 3.3. O modo incremental executa um ataque onde o nível aumenta de acordo com a frequência passada pelo usuário.

Tabela 3.3: Intensidade por pacotes/segundo

| **Level** | **Pacotes/segundo** |
|---|---|
| 1 | 1 |
| 2 | 10 |
| 3 | 100 |
| 4 | 1000 |
| 5 | 10000 |
| 6 | 100000 |
| 7 | 1000000 |
| 8 | 10000000 |
| 9 | 100000000 |
| 10 | 1000000000 |

O Injector é responsável pela injeção dos pacotes. Assim, ele é responsável pela criação e destruição das threads injetoras.

O Logger faz o log da injeção que pode ser apresentado em terminal ou salvo em arquivo. Esse log é exemplificado na Figura 3.3.

O módulo foi implementado em src/netuno por três arquivos como representado na Tabela 3.1. A parte Controller foi implementada no arquivo netuno.c. As outras partes são implementadas em arquivos com seu nome.

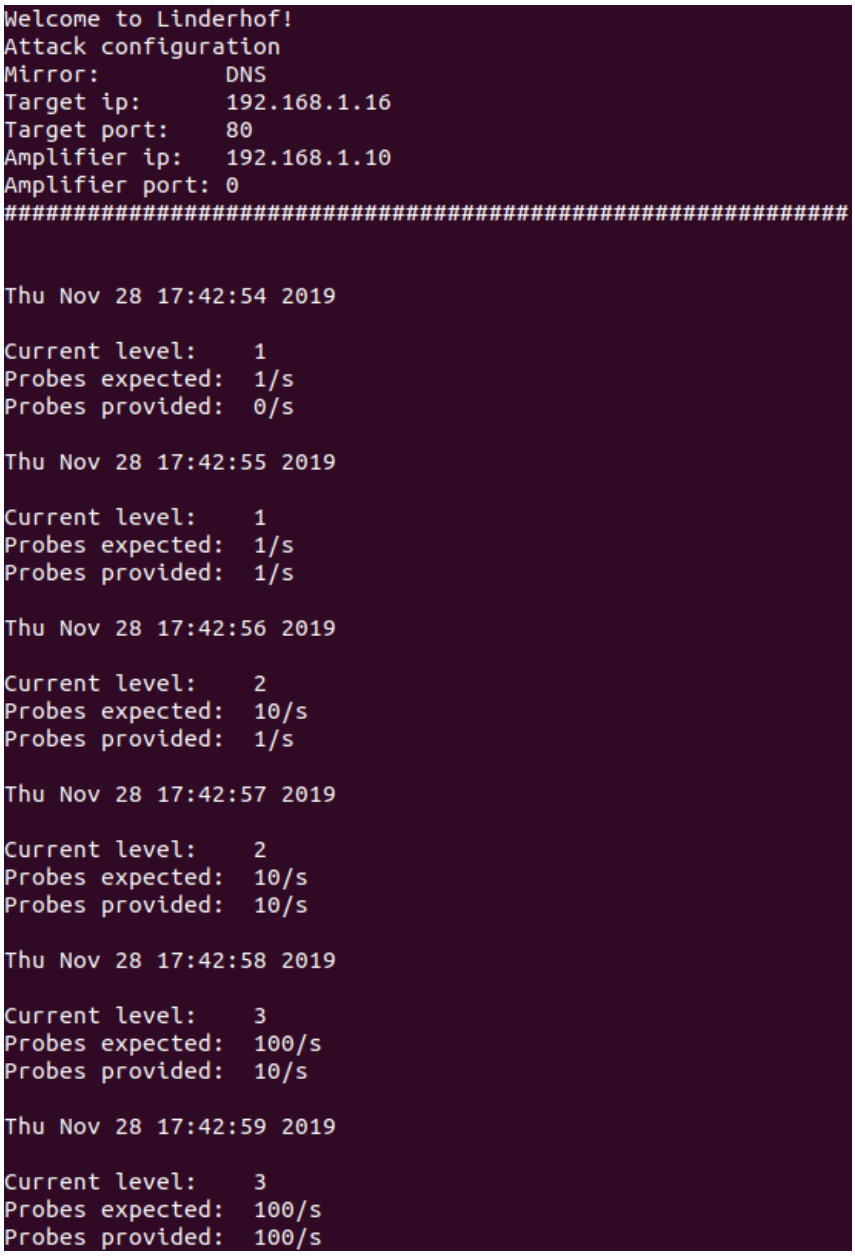


Figura 3.3: Exemplo de log para um ataque incremental com frequência de 2 segundos e duração de 6 segundos.

## 3.4 Diagnóstico

Foram identificados inconsistências na arquitetura original e na implementação dessa arquitetura. Além, disso a arquitetura original é complexa e a tradução dessa para imple-

mentação é confusa. Então, uma nova versão do Linderhof deve ser criada para resolver esses problemas e unificar o código.

### 3.4.1 Inconsistências Arquiteturais

A arquitetura descrita na Figura 3.1 é confusa e complexa. O nome dado a cada módulo não representa sua funcionalidade, o que complica a legibilidade da ferramenta.

A nomenclatura também apresentou problemas de consistência. O módulo Mirror Library é chamado Hall of Mirrors na documentação e implementação. Já o módulo Core é chamado Linderhof na implementação.

Por definição, toda interação com o usuário devia ser concentrada no módulo da interface. Porém, o submódulo Logger, que é responsável pelo *output* do programa, faz parte do módulo Netuno.

Outra inconsistência é a funcionalidade de validação do *draft* que foi implementada como validação dos parâmetros de *input*, assim, ela deve ser concentrada na interface e não no Módulo Core.

Quando comparados com sua implementação, os submódulos Executer e Planner foram considerados desnecessários, visto que foram implementados em poucas linhas de código, onde existem apenas duas funções para o Executor e uma para o Planner, e são chamados em sequência pelo Manager.

### 3.4.2 Inconsistências na Implementação dos Módulos

A organização do código-fonte, representada na Tabela 3.1, não segue a arquitetura proposta. No código, o submódulo Mirror Library é quem chama o Netuno, mas na arquitetura essa responsabilidade é do submódulo Executor.

A implementação da execução do programa também representa um problema. Já que o programa começa pela execução do Módulo Core que, por sua vez, chama a interface, diferindo do design apresentado.

A implementação dos mirrors, apresentada na Tabela 3.2, diverge em nomenclatura e quantidade de arquivos e apresenta *headers* fora da pasta include. Assim, deve ser padronizada.

Outro grande problema é a implementação do mirror *Simple Service Discovery Protocol* (SSDP). Já que ele implementa funcionalidades que não condizem com a arquitetura, como um scanner de refletores. No geral, esse mirror não segue a padronização da ferramenta e se comporta como um programa completamente diferente dos outros mirrors.

O código para adicionar as Linux *capabilities* dentro do Módulo Core é reproduzido no arquivo de automação da compilação, chamado build.sh, e não cumpre sua funcionalidade. Assim, ele não é relevante para a execução da ferramenta.

# Capítulo 4

# Execução e Resultados

O trabalho de refatoração/reestruturação tem como objetivo: simplificar a relação dos usuários futuros com a ferramenta; ajudar nos esforços de manutenção; facilitar a inclusão de novas funcionalidades; e permitir a inclusão e a modificação de mirrors de maneira padronizada e simples. Assim, para alcançar esses objetivos, foram tomadas as decisões de melhorar a qualidade do código com o esforço de refatoração, simplificar a arquitetura com esforço de reestruturação e documentar a ferramenta. A seguir descrevemos como esses objetivos foram atingidos.

## 4.1 Metodologia

O código-fonte da ferramenta estava hospedado na plataforma GitHub e, ao longo de seu desenvolvimento, foi distribuído em vários *forks*. Assim, primeiro é necessário consolidar o código-fonte. Na sua consolidação, o código foi exportado para o espaço do Laboratório de Ciber Segurança hospedado na GigaCandanga [36], garantindo o acesso restrito à ferramenta. O Linderhof está, então, sobre guarda do Laboratório de Ciber Segurança e o acesso ao código-fonte é controlado. Para o gerenciamento do código e da documentação, foi escolhida a plataforma Gitlab.

Como a legibilidade do código estava comprometida, graças à quantidade de código morto, primeiro a refatoração foi feita. Após a refatoração, a arquitetura foi simplificada e melhorada com a reestruturação. A documentação da implementação foi elaborada em conjunto com a reestruturação. Por fim, a documentação auxiliar foi elaborada.

## 4.2 Refatoração

A refatoração foi feita arquivo por arquivo. O código morto encontrado nesses arquivos foi excluído e os *bad smells* (para mais detalhes consulte a Tabela 2.2) resolvidos. Os

comentários que não trazem informação, como o comentário “ magic happens ” na Figura 4.3, também foram retirados. Todos os arquivos que compões o código-fonte foram examinados.

O código original continha diversas estruturas de código legado ou abandonado em comentário. Assim, para melhorar a legibilidade, o código morto foi retirado. Um grande exemplo de código morto encontrado nos arquivos é a estrutura Oxynet, que tinha o objetivo de ser uma segunda opção de interface, mas nunca foi implementada por completo. Assim, espalhado em seis arquivos, havia pedaços de código da estrutura que não eram executados ou até compilados, como no exemplo mostrado na Figura 4.1.

A Tabela 3.1 também exemplifica o código morto encontrado na implementação original, com todos os arquivos e pastas em vermelho não sendo utilizados na ferramenta. A Figura 4.2 é outro exemplo de código morto encontrado, já que a função coapSetValue não só não tinha corpo, como também nunca era chamada. Já a Figura 4.3 exemplifica um tipos muito comum de código morto, que são os códigos comentados.

```
26  #ifdef ORYXNET
27  static bool validateCmd( CommandPkt p_cmd )
28  {
60  }
61
62  static CommandPkt * packetToCmd( char * p_pkt )
63  {
80  }
81
82
83  static void serverInitializer( )
84  {
105 }
106
107 static ClientAddr * waitForClient()
108 {
136 }
137
138 static LhfDraft * getAttackDraftFromCmd( CommandPkt p_cmd )
139 {
152 }
153
154 static CommandPkt * getCommand( ClientAddr p_addr )
155 {
179 }
180
181 static void * commandHandler( void * p_addr )
182 {
214 }
215
216 void OryxNet()
217 {
225 }
226
227 void CloseOryxNet()
228 {
229     CloseSocket(serverSock);
230 }
231 #endif
```

Figura 4.1: Exemplo de código morto no arquivo oryx.c

```
17  void coapSetValue( AttackPlan * p_atkData )
18  {
19      //use for finding best URI
20  }
21
```

Figura 4.2: Exemplo de código morto no arquivo manager.c do mirror CoAP

Outro esforço da refatoração foi a correção de *bad smells* (veja Tabela 2.2 para detalhes) no código. Um exemplo de *bad smell* encontrado está na Figura 4.4a onde o comando switch/case é desnecessário, já que, em todos os mirrors menos o memcached, draft->type só pode assumir um tipo. Também foram encontradas algumas estruturas de dados desnecessárias, como na Figura 4.4b, que exemplifica a struct AtackPlan cujas

```
// magic occurs
ntp_header->rm_vn_mode=0x17;       //Sets the response bit to 0, More bit to 0, Version field to 2, Mode field to 7
ntp_header->auth_seq=0x00;    /* key, sequence number */
ntp_header->implementation=0x03;   //Sets the implementation to 3
ntp_header->request=0x2a;

//  // rm_vn_mode;     /* response, more, version, mode */
// ntp_header->auth_seq=0x00;    /* key, sequence number */
// ntp_header->err_nitems=0x0000;    /* error code/number of data items */
// ntp_header->mbz_itemsize=0x0000;    /* item size */
// ntp_header->tstamp=0x00000000;         /* time stamp, for authentication */
// ntp_header->keyid=0x0000;        /* encryption key */
```

Figura 4.3: Exemplo de código morto no arquivo ntpforge.c do mirror NTP

```
int  ExecuteNtpMirror( void *p_draft )
{
    AttackPlan *plan;
    LhfDraft *draft = (LhfDraft *)p_draft;
    switch( draft->type )
    {
        case NTP:
        default:
            plan = createAttackDataNTP( draft );
            break;
    }
    executeAttackNtp(plan);
    return 0;
}
```

(a) Uso desnecessário do comando swith-case.

```
typedef struct {
 Packet * getPacket;
 Packet * setPacket;
 LhfDraft *draft;
}AttackPlan;
```

(b) Bad smell em variáveis não usadas.

Figura 4.4: Exemplos de bad smells.

variáveis getPacket e setPacket eram apenas necessárias no mirror memcached, apesar da estrutura estar em todos os mirrors. Assim, nos outros protocolos, as variáveis getPacket e setPacket não eram usadas.

A Figura 4.5 também exemplifica uma estrutura desnecessária na função executeCoapAttack, que só era chamada pela função ExecuteCoapMirror. Como a modalidade de log em arquivo foi retirada, a linha de código que declara a variável fileName é desnecessária. Assim, a função executeCoapAttack executava apenas a chamada para a função StatNetunoInjector, ou seja, a chama da função executeCoapAttack na função ExecuteCoapMirror pode ser substituída pela chamada da função StartNetunoInjector. Essa situação estava sendo reproduzida em todos os mirrors.

Outro exemplo de *bad smell* foi encontrado no comportamento dos refletores. A ferramenta tinha suporte para apenas um refletor, cujo IP era passado por parâmetro. Porém, no código, eram *hard coded* três refletores e o parâmetro era ignorado, como mostrado na Figura 4.6. Assim, foi implementado o suporte para mais de um refletor com o uso de um arquivo .txt. O número de injetores era *hard coded* no código, setado para 10, assim, como isso caracteriza um *bad smell*, foi implementado um número variável de injetores.

Foram encontrados alguns *bugs*, como no mirror DNS, cuja execução poderia resultar em *buffer overflow*. Também foi encontrada e corrigida uma mal formação nos pacotes do mesmo mirror. Outro exemplo de bug foi encontrado no comportamento do mirror NTP, que enviava um pacote a mais durante o ataque.

```
void executeCoapAttack( AttackPlan * atkData )
{
    char *fileName = (atkData->draft->logfile[0] == '\0') ? NULL : atkData->draft->logfile;

    StartNetunoInjector( atkData->getPacket, atkData->draft->level, atkData->draft->timer, atkData->draft->incAttack, fileName);
}

int  ExecuteCoapMirror( void *p_draft )
{
    AttackPlan *plan;
    LhfDraft *draft = (LhfDraft *)p_draft;


    plan = createAttackData( draft );

    executeCoapAttack(plan);
    return 0;
}
```

Figura 4.5: A função executeCoapAttack é um *bad smell*

A porta de origem do pacote era constante e definida por mirror. Assim, foi acrescentada a aleatoriedade na escolha da porta de origem do pacote. A porta é mantida para um mesmo nível e incrementada na mudança de nível. Esse comportamento foi escolhido para auxiliar na análise da dinâmica do ataque.

Por fim, o tratamento de erro na ferramenta estava presente, porém as mensagens de erro eram, em sua maioria, pouco explicativas e até confusas. As mensagens existentes foram adequadas e o tratamento de erro foi expandido ao longo do código.

```
//FILE *listReflect = fopen("refletors.csv", "r");
#define NN 3
//struct sockaddr in saddr[MAXINJECTORS];
char addrs[NN][16] = {"192.168.1.10", "192.168.1.11", "192.168.1.12"};
// char addrs[3][16] = {"192.168.0.63", "192.168.0.63", "192.168.0.63"};
void *pack_rot[NN] = {0};
```

Figura 4.6: Exemplo de *bad smell* no *handling* dos refletores.

## 4.3 Reestruturação

A definição de uma nova arquitetura é necessária para resolver as inconsistências e os problemas encontrados. Essa arquitetura é descrita a seguir.

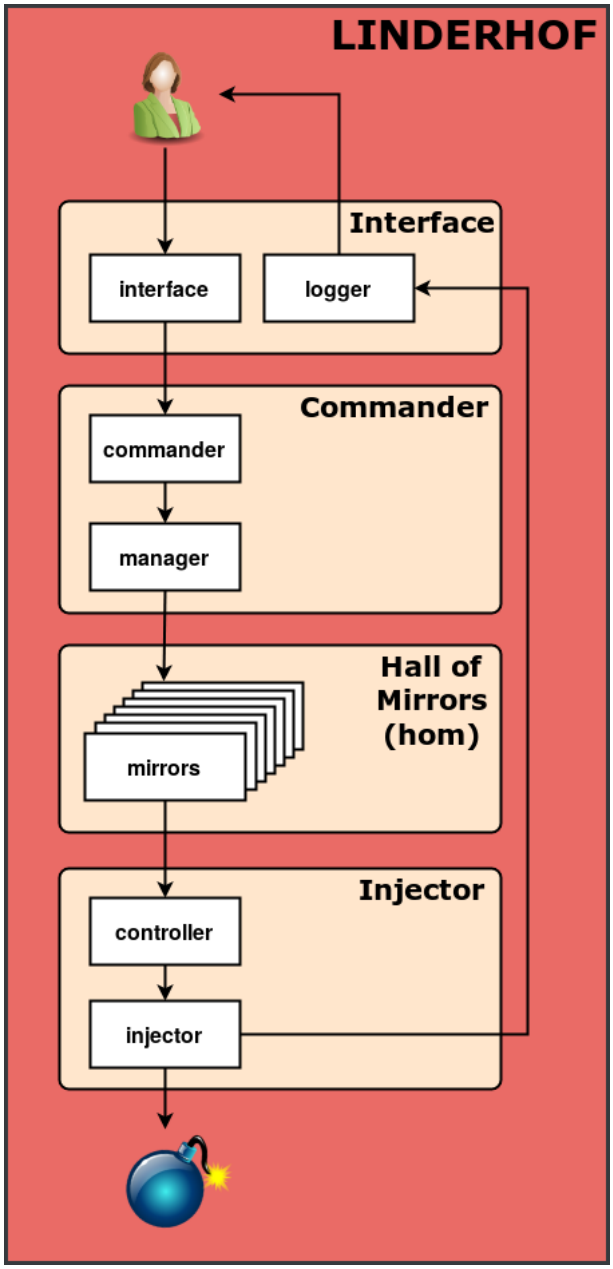


Figura 4.7: Nova Arquitetura do Linderhof

Tabela 4.1: Organização da pasta src depois da refatoração.

| Caminho | Arquivos | Descrição |
|---|---|---|
| commander | commander.c<br>manager.c | Commander |
| hom/<mirror> | <mirror>.c<br><mirror>forge.c | Commander |
| common | blacksmith.c<br>error.c<br>list_reflector.c<br>memutils.c<br>netio.c<br>pthreadHelper.c<br>timeHelper.c | Arquivos usados em mais de um módulo |
| include/commander<br>include/common<br>includer/injector<br>include/interface | *.h | Headers |
| include/hom/<mirror> | <mirror>.h<br><mirror>forge.h | Headers dos mirrors |
| include | draft.h | Implementação da estrutura draft e estruturas auxiliares |
| injector | controller.c<br>injector.c | Injector |
| interface | cliparser.c<br>configparser.c<br>interface.c<br>loger.c | Interface |

### 4.3.1 Nova Arquitetura

A nova arquitetura, representada na Figura 4.7, é uma simplificação da arquitetura antiga. É dividida em quatro módulos base: **Interface**, **Commander**, **Hall of Mirrors** e **Injector**. Os módulos Interface, Commander e Injector são derivados, respectivamente, dos antigos módulos Oryx, Core e Netuno. A nova arquitetura é descrita a seguir.

### Interface

O módulo da Interface é responsável por toda interação software-usuário. Assim como no antigo Módulo Oryx, é montado um rascunho do ataque fazendo uso da estrutura de dados **draft**, no submódulo interface, onde estão presentes todas as informações necessárias para montar o ataque. Esse draft é encaminhado para o módulo Commander. A parte logger do módulo é responsável pelas informações de saída do programa. Essas informações dizem respeito à montagem principal do ataque e ao processo de injeção.

A implementação do módulo pode ser vista na Tabela 4.1 em src/interface. Os antigos arquivos interface.c e oryx.c (para mais detalhes consultar a Tabela 3.1), foram consolidados em um arquivo chamado interface.c. Os arquivos cliparser.c e configparser.c são a implementação de parsers, sendo o cliparser um parser dos argumentos lidos em linha de comando e o configparser responsável pelo arquivo de configuração. O arquivo cliparse.c, antes na pasta common, foi transferido para pasta interface e o arquivo configparser.c foi incluído.

### Commander

O módulo Commander é dividido em duas partes: o commander e o manager. A parte commander é responsável por inicializar a ferramenta. A inicialização da ferramenta seta os *handlers* de sinal e as funções de erro. A parte commander é chamada pelo módulo da Interface e, após sua execução, chama a parte manager.

A parte manager é o resultado da junção dos antigos submódulos Planner e Executor. Assim, ela é responsável pela chamada e execução do ataque que mudam de acordo com o mirror escolhido. Isso é feito alocando a função de chamada do mirror no draft.

A implementação desse módulo pode ser vista na Tabela 4.1 em src/commander. Os antigos arquivos planner.c e executer.c (representados na Tabela 3.1) foram consolidados no arquivo manager.c de acordo com a nova arquitetura. O antigo arquivo linderhof.c foi renomeado para commander.c.

### Hall Of Mirrors

Algumas funcionalidades que irão compor as versões futuras do Linderhof pedem que o Hall of Mirrors seja um módulo separado na arquitetura, já que será necessário acessá-lo fora do fluxo comum do ataque e outros fluxos de ataque que não irão acessá-lo serão criados. Assim, a parte manager chama o módulo Hall of Mirrors. Nele, todos os protocolos implementados pela ferramenta podem ser encontrados na forma de mirrors. Em cada mirror é forjado o pacote que será passado para o Injector e enviado no ataque. Dependendo da escolha do protocolo, o pacote muda.

A implementação desse módulo pode ser vista na Tabela 4.1 em src/hom. A implementação dos mirrors foi padronizada, de acordo com a Tabela 4.1, sendo que <mirror> deve ser substituído pela sigla do protocolo em letra minúscula.

**Injector**

O módulo Injector é responsável pela injeção dos pacotes. Ele é a reestruturação do antigo módulo Netuno. Assim, o módulo Hall of Mirrors chama a parte controller, que controla todos os aspectos da injeção. O controller chama a parte injector para iniciar as *threads* injetoras e gerar a taxa de injeção de cada uma.

A parte injector é responsável pela criação e destruição das *threads* injetoras. A cada *thread* injetora, é relacionado um *bucket* que diz respeito à quantidade de pacotes que devem ser enviados. É pela atualização dos *buckets* que o controller gerencia a taxa de injeção de acordo com a intensidade do ataque desejada.

A implementação desse módulo pode ser vista na Tabela 4.1 em src/injector. O antigo arquivo netuno.c (representado na Tabela 3.1) foi renomeado para controller.c.

### 4.3.2 Funcionalidades

Esse trabalho, em conjunto com o trabalho do aluno de graduação Matheus Oliveira Vieira e o trabalho do mestrando Alan Tamer Vasques, compõe a versão 1.0.0 do Linderhof.

Nessa versão, foi acrescentado o suporte para o protocolo IPv6. Também foi implementado o suporte para um arquivo de configuração do ataque pelos parâmetros de linha de comando. Os mirrors foram revisados e alguns parâmetros específicos aos protocolos foram incluídos na ferramenta. O mirror SSDP foi implementado novamente e o mirror SNMP foi acrescentado. Esses esforços fazem parte do trabalho do aluno Matheus Oliveira Vieira.

O mecanismo de automação da compilação foi revisado e o suporte para depuração foi acrescentado. O suporte ao arquivo de Log foi abandonado. E a saída do programa foi remodelada para facilitar o processo de análise do ataque. A Figura 4.8 representa a nova saída. Como já mencionado, o suporte para mais de um refletor foi implementado fazendo uso de um arquivo.

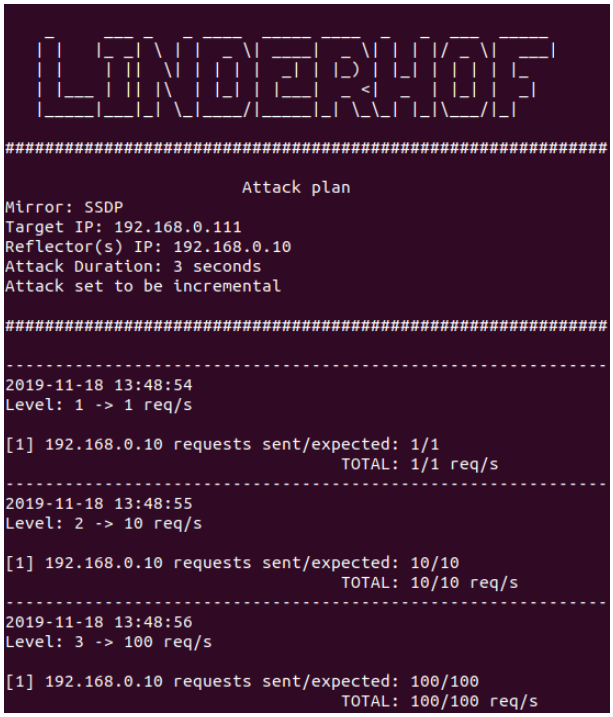


Figura 4.8: Exemplo de saída do Linderhof.

A ferramenta foi testada no trabalho do mestrando Alan Tamer Vasques que também foi responsável pela gerência do projeto. A Tabela 4.2 apresenta um resumo de todas as funcionalidades da ferramenta.

Tabela 4.2: Resumo das Funcionalidades

| Nome | Descrição | Histórico | Módulo |
|---|---|---|---|
| Arquivo de Configuração | Ajuda na personalização do ataque | Adicionada | Interface |
| Arquivo de Log | Faz o log em arquivo | Retirada | N/A |
| Duração | Tempo de duração do ataque | Mantida | Injector |
| Mirrors | Protocolos a serem explorados no ataque | Alterada | Hall of Mirrors |

Tabela 4.2 – Continuação

| Nome | Descrição | Histórico | Módulo |
|---|---|---|---|
| Modo Agressivo | Envia o número de pacotes do nível para cada refletor | Adicionada | Injector |
| Modo Flood | Envia o máximo de pacotes possível | Adicionada | Injector |
| Modo Incremental | Aumenta o nível do ataque de acordo com a informação passada pelo usuário | Mantida | Injector |
| Múltiplos Refletores | Ataque com mais de um refletor | Adicionada | Interface Injector |
| Nível | Diz respeito a quantidade de pacotes enviada de acordo com a Tabela 3.3 | Mantida | Injector |
| Parâmetros do Ataque | Permite que o usuário defina suas preferências | Mantida | Interface |
| Parâmetros para o Mirror | Permite a passagem de argumentos de linha de comando para o mirror | Adicionada | Hall of Mirrors |
| Suporte IPv4 e IPv6 | Suporta as duas versões do IP | Modificada | Interface Injector |

Foram acrescentados também, pelo trabalho do Matheus, dois novos modos de ataque. No modo de ataque *default* é gerado um número fixo de pacotes por nível e esses pacotes são distribuídos igualmente entre cada refletor. O modo *default* é exemplificado pela Figura 4.9. No modo de ataque incremental, a ferramenta aumenta o nível do ataque (para mais detalhes consulte a Tabela 3.3) de acordo com a frequência passada pelo usuário.

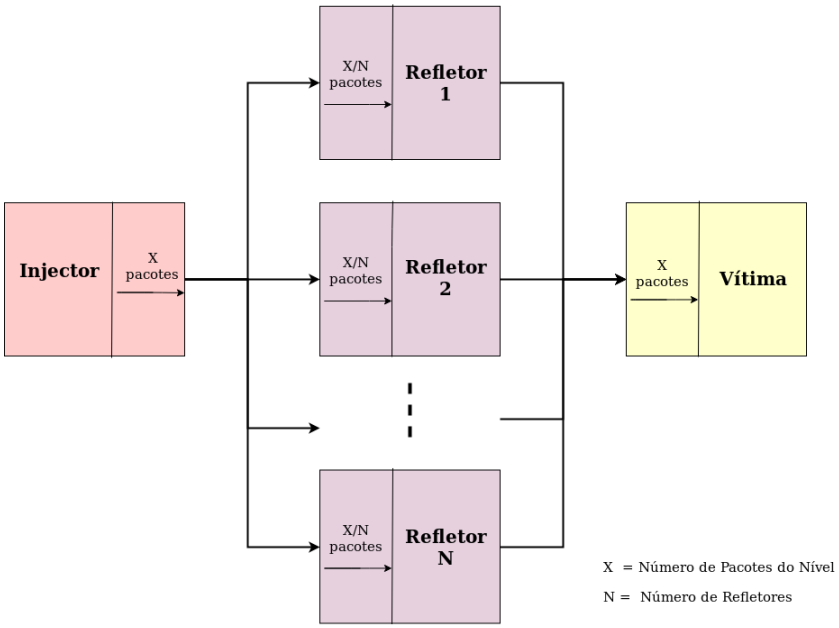


Figura 4.9: Modo de ataque *default*.

O novo modo agressivo extrapola o nível do ataque para os refletores. Ou seja, cada refletor recebe a quantidade total de pacotes desejada para aquele nível. O modo agressivo é exemplificado na Figura 4.10. Já para o modo flood, não existem níveis. A ferramenta entra em loop e envia o máximo possível de pacotes para todos os refletores.

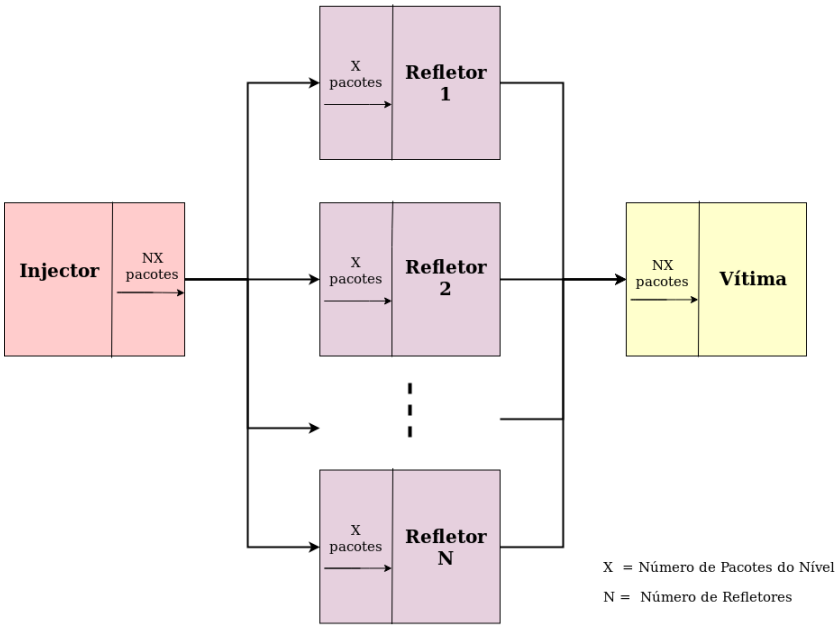


Figura 4.10: Modo de ataque agressivo.

Todos os argumentos aceitos pelo programa por linha de comando são listados na Figura 4.11

### 4.3.3 Implementação

O Hall of Mirror é composto por 6 protocolos: *Constrained Application Protocol* (CoAP), *Domain Name System* (DNS), Memcached, *Network Time Protocol* (NTP), *Simple Service Discovery Protocol* (SSDP) e *Simple Network Management Protocol* (SNMP). Sua implementação foi padronizada e, junto com a organização do código-fonte, pode ser vista na Tabela 4.1.

Durante o processo de refatoração foram encontrados problemas na estrutura da implementação. Um desses problemas é o arquivo signalhandler (src/common representado na Tabela 3.1) que implementa somente uma função, que, por sua vez, é utilizada apenas uma vez pelo Módulo Core na arquitetura antiga. A função implementada nesse arquivo foi realocada para o arquivo commander.c (src/commander na Tabela 4.1).

Já o arquivo cliparser, localizado na pasta src/common (Tabela 3.1, é usado apenas pelo Módulo Oryx. Como a pasta common deve conter arquivos usados por mais de um módulo, esse arquivo foi realocado para a pasta src/interface (Tabela 4.1).

O arquivo capabilityHelper, localizado na pasta src/common (Tabela 3.1, fazia parte do código para adicionar as Linux *capabilities* que foi excluído da ferramenta. Por fim, o arquivo venus.h (src/include na Tabela 3.1) foi renomeado para draft.h.

| Nome | Opção Longa | Opção curta | Opção Obrigatória | Argumento Obrigatório | Valor Padrão | Descrição |
|---|---|---|---|---|---|---|
| ARG_MIRROR | mirror | m | ✔ | ✔ | Não possui | Espelho que será utilizado no ataque |
| ARG_TARGET_IP | target | t | ✔ | ✔ | Não possui | IP da vítima |
| ARG_REFLECTOR_IP | reflector | r | ✔ | ✔ | Não possui | IP do refletor |
| ARG_REFLECTOR_PORT | refleport | p | ✘ | ✔ | Porta padrão do mirror | Porta do refletor |
| ARG_TARGET_PORT | targport | g | ✘ | ✔ | Aleatória (40000 - 60000) | Porta do vítima |
| ARG_LEVEL | level | l | ✘ | ✔ | 1 | Nível do ataque |
| ARG_DURATION | duration | d | ✘ | ✔ | Infinito | timer em segundos |
| ARG_FLOOD | flood | f | ✘ | N/A | Desativado | Ativa modo flood |
| ARG_INCREMENT | inc | i | ✘ | ✔ | Desativado | Ativa modo incremental |
| ARG_AGGRESSIVE | aggressive | a | ✘ | N/A | Desativado | Ativa modo agressivo |
| ARG_CONFIG | config | c | ✘ | ✘ | N/A | Arquivo de configuração |
| ARG_DNS_DOMAIN | domain-name | D | ✘ | ✔ | ddos.dns.com | Parâmetro específico do mirror DNS |
| ARG_SSDP_UPNP_VERSION | upnp-version | V | ✘ | ✔ | UPNP_V1_0 | Parâmetro específico do mirror SSDP |
| ARG_SSDP_UNICAST | unicast | U | ✘ | N/A | N/A | Parâmetro específico do mirror SSDP |
| ARG_SNMP_COMMUNITY_STRING | comunity-string | C | ✘ | ✔ | public | Parâmetro específico do mirror SNMP |
| ARG_SNMP_MAX_REPETITIONS | max-repetitions | R | ✘ | ✔ | 2000 | Parâmetro específico do mirror SNMP |
| ARG_COAP_SZX | szx | Z | ✘ | ✔ | 6 | parâmetro específico do mirror COAP |
| ARG_COAP_URI_PATH | uri-path | P | ✘ | ✔ | .well-known/core | Parâmetro específico do mirror COAP |

Figura 4.11: Argumentos aceitos por Linha de Comando.

## 4.4 Documentação

Na ferramenta Linderhof, a documentação foi separada em duas partes: a documentação de código-fonte e a documentação wiki. Os estilos de documentação, assim como quais arquivos compõem a documentação, foram escolhidos levando em consideração os usuários futuros da ferramenta. Assim, o objetivo da documentação é ser completa, mas

pouco extensa. O conceito de código como documentação foi referência nesse processo. A documentação foi produzida seguindo os padrões da documentação *open-source*, graças a natureza colaborativa do código. A documentação wiki complementa a documentação do código-fonte com guias de uso, explicações de instalação e diagramas de relacionamento.

A documentação do código-fonte foi concentrada nos arquivos de *header* para melhor legibilidade. Explicações adicionais necessárias para o entendimento das funções e estruturas presentes apenas no arquivo fonte foram comentadas no próprio arquivo fonte. A documentação por comentário foi feita no padrão Doxygen [37], exemplificado na Figura 4.12. Ele foi escolhido por ser o padrão mais comum nos projetos ao longo da graduação na universidade, além de ser um padrão suportado por muitas *integrated development environments* (IDEs). A documentação do código foi gerada fazendo uso do Doxygen e anexada à documentação da ferramenta, montando um *reference file*.

```
/** @brief gets the size of an injection bucket
 *
 *  @param injector_idx injector id
 *  @param level
 *  @param aggressive_mode flag
 *  @return size of the bucket
 */
int getBucketSize(int injector_idx, uint8_t level, uint8_t aggressive_mode);
```

Figura 4.12: Exemplo de documentação do código-fonte.

Como o estilo de documentação seguido foi o dos projetos *open source*, foi elaborado um arquivo README.md apresentado no Apêndice A. Um exemplo da documentação auxiliar presente na wiki do projeto pode ser encontrado no Apêndice B.

A usabilidade da plataforma também entrou no esforço de documentação. As mensagens explicativas durante a compilação e execução foram enriquecidas, com melhorias feitas na mensagem do parâmetro –help e a implementação de mensagens explicativas para o arquivo build.sh.

### 4.4.1 Resultados

A implementação original era composta por 76 arquivos com 3.295 linhas de código efetivas. A nova implementação contém 57 arquivos com 3.034 linhas de código efetivas. Ao todo foram acrescentadas 20 novas funções, excluídas 40 funções e alteradas 26 funções. A quantidade de linhas de código morto excluídas totalizou 863. A Tabela 4.3 apresenta um resumo da nova arquitetura, que é resultado dos esforços de reestruturação, com as informações das responsabilidades e funcionalidades implementadas por cada módulo, além da interação entre os módulos, ou seja, por qual módulo ele é chamado e qual o módulo que ele chama. Um resumo das funcionalidades citadas pode ser encontrado na Tabela 4.2.

A documentação adicional em formato wiki totalizou 543,4 kB com 7 arquivos e 8 imagens, sendo que, quando transformada em pdf, resultou em 12 páginas de documentação adicional. A documentação automática gerada pelo Doxygen totalizou 179 páginas em pdf. No total foram escritas 621 linhas de comentário para documentação.

Tabela 4.3: Resumo da Nova Arquitetura

| **Módulo** | **Responsabilidades** | **Funcionalidades** | **Chamado** | **Chama** |
|---|---|---|---|---|
| Interface | Interação com o usuário e montagem do draft | Suporte IPv4 e IPv6<br>Arquivo de configuração<br>Parâmetros do Ataque<br>Múltiplos Refletores | Usuário (main) | Commander |
| Commander | Inicialização do Linderhof e do ataque | - | Interface | Hall of Mirrors |
| Hall of Mirrors | Forge de Pacotes | Mirros<br>Parâmetros para o Mirror | Commander | Injector |
| Injector | Injeção de Pacotes | Nível<br>Modo Agressivo<br>Modo Flood<br>Modo Incremental<br>Duração<br>Múltiplos Refletores<br>Suporte IPv4 e IPv6 | Hall of Mirrors | Interface |

# Capítulo 5

# Conclusão e trabalhos futuros

Essa monografia apresentou as técnicas de reestruturação e refatoração para a manutenção da ferramenta Linderhof.

Primeiro foram apresentados os princípios de um ataque de negação de serviço, implementado pela ferramenta, e as técnicas de reestruturação e refatoração. Também foram apresentados os princípios da documentação de software.

Em seguida, foi apresentada a ferramenta com sua arquitetura original e sua implementação. Foi apresentado um diagnóstico juntamente com novos requisitos que motivaram os processos de refatoração, reestruturação e documentação da ferramenta.

No processo de refatoração, o código morto foi retirado do código, foram identificados e resolvidos diversos *bad smells*, a funcionalidade do arquivo de refletores foi implementada e o tratamento de erro foi expandido pela ferramenta. No processo de reestruturação, foram relatados os problemas encontrados na arquitetura original e uma nova arquitetura foi apresentada e implementada, resolvendo os problemas identificados.

E, por fim, foi elaborada uma documentação da ferramenta. Essa documentação conta com a documentação do código e com uma documentação auxiliar voltada para os futuros usuários.

A refatoração e a reestruturação melhoraram a qualidade da ferramente e a tornaram menos complexa. O código da nova implementação é mais legível que o código original e, com o apoio da documentação, a ferramenta é mais facilmente compreendida e estudada. Assim, a inclusão de novas funcionalidades foi simplificada. Como culminação desse processo, ocorreu o lançamento da versão 1.0.0 da ferramenta.

## 5.1 Trabalhos Futuros

Os seguintes trabalhos futuros são propostos:

- **Implementação de um scanner de refletores:** Atualmente a ferramenta Linderhof pede que os IPs dos refletores usados no ataque sejam informados por linha de comando. Porém, na realidade, para um ataque de negação de serviço, a identificação de refletores é crucial. Assim, a implementação de um scanner de refletores na ferramenta a tornaria mais realista.

- **Implementação de outros tipos de ataque de negação de serviço**: Diferentes variedades de ataques de negação de serviço estão se tornando cada vez mais populares. Assim, implementar diferentes tipos de ataque na ferramenta a tornaria mais completa. A inclusão de ataques de *flooding*, *carpet bombing* e *pulse attacks* é recomendada.

- **Inclusão de uma interface gráfica**: Atualmente, a ferramenta faz uso de um grande número de *flags* para passagem de parâmetro por linha de comando, o que diminui a usabilidade da ferramenta. Assim, uma interface gráfica diminuiria o esforço de uso das funcionalidades.

- **Novos mirrors**: Ainda existem protocolos explorados por ataques de negação de serviço que não foram incluídos na ferramenta.

# Referências


[1] *Reference documentation*. https://docs.docker.com/reference/, 2019. ix, 17

[2] *Wiki:home*. https://github.com/otwcode/otwarchive/wiki, 2019. ix, 18

[3] *Netscout threat intelligence report: Dawn of the terrorbit era (finding from second half 2018)*. https://www.netscout.com/sites/default/files/2019-02/SECR_001_EN-1901%20-%20NETSCOUT%20Threat%20Intelligence%20Report%202H%202018.pdf. x, 6, 7

[4] Fowler, Martin e Kent Beck: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2ª edição, 2018, ISBN 0134757599. x, 8, 9, 10

[5] Medeiros, Tiago Fonseica: *Ataque distribuído de negação de serviço por reflexão amplificada usando simple network management protocol*. http://bdm.unb.br/handle/10483/11152, 2015. 1

[6] Hirata, Henrique Senoo: *Ataque de negação de serviço por reflexão amplificada usando simple service discovery protocol*. http://bdm.unb.br/handle/10483/22128, 2018. 1

[7] Miranda, Igor: *Ataque de negação de serviço por reflexão amplificada explorando memcached*, 2018. 1

[8] Vieira, Alexander André de Souza: *Ataque distribuído de negação de serviço por reflexão amplificada usando network time protocol*, 2019. 2

[9] Pereira, Pedro Henrique Morais: *Internet das coisas e seus riscos: uma análise da exploração de servidores coap como refletores de ataques de negação de serviço amplificados*, 2019. 2

[10] Riza, Ahmad, Rizaain Yusof, Nur Udzir e Ali Selamat: *Systematic literature review and taxonomy for ddos attack detection and prediction*. International Journal of Digital Enterprise Technology, 1:292, janeiro 2019. 4

[11] Specht, Stephen M. e Ruby B. Lee: *Distributed Denial of Service: Taxonomies of Attacks, Tools and Countermeasures*. International Workshop on Security in Parallel and Distributed Systems, (9):543–550, 2004, ISSN 15384047. 4

[12] Peng, Tao, Christopher Leckie e Kotagiri Ramamohanarao: *Survey of network-based defense mechanisms countering the dos and ddos problems*. ACM Comput.

Surv., 39(1), abril 2007, ISSN 0360-0300. http://doi.acm.org/10.1145/1216370.1216373. 4, 5

[13] *What is a ddos attack?* https://www.cisco.com/c/en/us/products/security/what-is-a-ddos-attack.html, 2019. 5

[14] Zargar, S. T., J. Joshi e D. Tipper: *A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks*. IEEE Communications Surveys Tutorials, 15(4):2046–2069, Fourth 2013. 7

[15] Ferguson, P. e D. Senie: *Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing*. 2000. 7

[16] Jin, Cheng, Haining Wang e Kang G. Shin: *Hop-count filtering: An effective defense against spoofed ddos traffic*. Em *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, páginas 30–41, New York, NY, USA, 2003. ACM, ISBN 1-58113-738-9. http://doi.acm.org/10.1145/948109.948116. 7

[17] Limkar, Suresh e Rakesh Kumar Jha: *An effective defence mechanism for detection of ddos attack on application layer based on hidden markov model*. Em Satapathy, Suresh Chandra, P. S. Avadhani e Ajith Abraham (editores): *Proceedings of the International Conference on Information Systems Design and Intelligent Applications 2012 (INDIA 2012) held in Visakhapatnam, India, January 2012*, páginas 943–950, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg, ISBN 978-3-642-27443-5. 7

[18] Peraković, D., M. Periša, I. Cvitić e S. Husnjak: *Artificial neuron network implementation in detection and classification of ddos traffic*. Em *2016 24th Telecommunications Forum (TELFOR)*, páginas 1–4, Nov 2016. 7

[19] Stevanovic, Dusan, Natalija Vlajic e Aijun An: *Detection of malicious and non-malicious website visitors using unsupervised neural network learning*. Applied Soft Computing, 13(1):698 – 708, 2013, ISSN 1568-4946. http://www.sciencedirect.com/science/article/pii/S1568494612003778. 7

[20] Peng, Tao, Christopher Leckie e Kotagiri Ramamohanarao: *Defending against distributed denial of service attacks using selective pushback*. Em *In Proceedings of the Ninth IEEE International Conference on Telecommunications (ICT*, páginas 411–429, 2002. 8

[21] Wang, X.: *Mitigation of ddos attacks through pushback and resource regulation*. Em *2008 International Conference on MultiMedia and Information Technology*, páginas 225–228, Dec 2008. 8

[22] Yau, D. K. Y., J. C. S. Lui e Feng Liang: *Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles*. Em *IEEE 2002 Tenth IEEE International Workshop on Quality of Service (Cat. No.02EX564)*, páginas 35–44, May 2002. 8

[23] Walfish, Michael, Mythili Vutukuru, Hari Balakrishnan, David Karger e Scott Shenker: *Ddos defense by offense*. ACM Trans. Comput. Syst., 28:1–54, março 2010. 8

[24] Mens, Tom, Serge Demeyer, Bart Du Bois, Hans Stenten e Pieter Van Gorp: *Refactoring: Current research and future trends*. Electronic Notes in Theoretical Computer Science, 82(3):483 – 499, 2003, ISSN 1571-0661. http://www.sciencedirect.com/science/article/pii/S1571066105826246, LDTA'2003 - Language descriptions, Tools and Applications. 8, 11

[25] Philipps, J. e B. Rumpe: *Refinement of information flow architectures*. Em *First IEEE International Conference on Formal Engineering Methods*, páginas 203–212, Nov 1997. 12

[26] Tokuda, Lance e Don Batory: *Evolving object-oriented designs with refactorings*. Automated Software Engineering, 8(1):89–120, Jan 2001, ISSN 1573-7535. https://doi.org/10.1023/A:1008715808855. 12

[27] Forward, Andrew e Timothy C. Lethbridge: *The relevance of software documentation, tools and technologies: A survey*. Em *Proceedings of the 2002 ACM Symposium on Document Engineering*, DocEng '02, páginas 26–33, New York, NY, USA, 2002. ACM, ISBN 1-58113-594-7. http://doi.acm.org/10.1145/585058.585065. 12

[28] Clements, Paul, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers e Reed Little: *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002, ISBN 0201703726. 13, 14

[29] Parnas, David Lorge: *Precise Documentation: The Key to Better Software*, páginas 125–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, ISBN 978-3-642-15187-3. https://doi.org/10.1007/978-3-642-15187-3_8. 13, 14

[30] Reeves, Jack W.: *What is software desing?* The C++ Journal, 2(2), 1992. http://www.developerdotstar.com/mag/articles/reeves_design.html. 15

[31] *The open source way*. https://opensource.com/open-source-way, 2019. 16

[32] *Starting an open source project*. https://opensource.guide/starting-a-project/, 2019. 16

[33] *A beginner's guide to writing documentation*. https://www.writethedocs.org/guide/writing/beginners-guide-to-docs/, 2019. 16

[34] *Running the endgame*. https://github.com/microsoft/vscode/wiki/Running-the-Endgame, 2019. 17

[35] Berglund, Erik e Michael Priestley: *Open-source documentation: In search of user-driven, just-in-time writing*. Em *Proceedings of the 19th Annual International Conference on Computer Documentation*, SIGDOC '01, páginas 132–141, New York, NY, USA, 2001. ACM, ISBN 1-58113-295-6. http://doi.acm.org/10.1145/501516.501543. 18

[36] *Gigacandanga*. https://www.gigacandanga.net.br/, 2019. 29

[37] *Doxygen*. http://www.doxygen.nl/, 2019. 42

# Apêndice A

# Documentação: arquivo README.md

# Linderhof

---

Linderhof é uma biblioteca de ataques distribuídos de negação de serviço por reflexão amplificada, com suporte a diversos protocolos (DNS, NTP, Memcached, SNMP, SSDP e CoAP).

# Instalação

Requisitos: - Cmake (versão 3.3 ou maior)

O arquivo **build.sh** é usado para a compilação do Linderhof. Os parâmetros aceitos pelo **build.sh** são:

```
-b Build
-g Generate build files
-d Generate build files for debugging
-h Show help
```

Assim, primeiro os build files devem ser gerados e então o build deve ser rodado. Um exemplo de compilação é:

```
$ ./build.sh -gb
```

O binário gerado (lhf) estará na pasta bin do projeto.

Caso o script não tenha sido rodado com privilégios administrativos, deve ser executado o comando abaixo para que deixe de ser necessário executar o lhf como superuser:

```
$ sudo setcap cap_net_admin,cap_net_raw+ep bin/lhf
```

# Execução

O Linderhof pode ser executado da seguinte forma:

```
$ bin/lhf -m <tipo_de_mirror> -t <IP_da_vítima> -r <IP_do_refletor | arquivo_com_lista_de_re
```

Exemplo:

```
$ bin/lhf -m ssdp -t 2001:db8::1 -r reflectors.txt
```

O tipo de mirror escolhido pode influenciar os argumentos necessários para a execução correta do programa. Todos os argumentos, tanto opcionais quanto obrigatórios, podem ser consultados na ajuda do programa:

```
$ lhf -h
```

# Apêndice B

# Documentação: documentação auxiliar

# Primeiros Passos

## Instalação

Requisitos: > **Cmake (versão 3.3 ou maior)**

Para compilar basta executar o script **build.sh**. O arquivo build.sh aceita os parâmetros:

```
-b Build
-g Generate build files
-d Generate build files for debugging
-h Show help
```

Assim, primeiro os arquivos build devem ser gerados e então o build deve ser rodado. Algumas verificações de dependências serão feitas e, caso não aconteça nenhum erro na compilação, o binário (lhf) estará disponível na pasta **bin** do projeto. Um exemplo para compilação é:

```
sudo ./build.sh -gb
-- The C compiler identification is GNU 7.4.0
-- The CXX compiler identification is GNU 7.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /tmp/linderhof/build

Scanning dependencies of target lhf
   ...
   <saída omitida>
   ...
[100%] Linking C executable ../bin/lhf
[100%] Built target lhf
```

Caso o script não tenha sido rodado com privilégios administrativos, o seguinte comando deve ser executado para que deixe de ser necessário executar o **lhf**

como superuser:

```
$ sudo setcap cap_net_raw+ep bin/lhf
```

## Uso

O Linderhof deve ser executado seguindo o exemplo abaixo, sendo as opções "-m", "-t" e "-r" obrigatórias:

```
$ bin/lhf -m <tipo_de_mirror> -t <IP_da_vítima> -r <IP_do_refletor | arquivo_com_lista_de_re
```

Exemplo: $bin/lhf -m ssdp -t 2001:db8::1 -r reflectors.txt

Argumentos para "tipo_de_mirror": - coap - dns - memcached_getset - memcached_stat - ntp - snmp - ssdp

O tipo de mirror escolhido pode influenciar as opções e argumentos necessários para a correta execução do programa. As outras opções e argumentos aceitos pelo programa estão relacionados abaixo:

| Nome | Opção Longa | Opção curta | Opção Obrigatória | Argumento Obrigatório | Valor Padrão | Descrição |
|---|---|---|---|---|---|---|
| ARG_MIRROR | mirror | m | ✔ | ✔ | Não possui | Espelho que será utilizado no ataque |
| ARG_TARGET_IP | target | t | ✔ | ✔ | Não possui | IP da vítima |
| ARG_REFLECTOR_IP | reflector | r | ✔ | ✔ | Não possui | IP do refletor |
| ARG_REFLECTOR_PORT | refleport | p | ✘ | ✔ | Porta padrão do mirror | Porta do refletor |
| ARG_TARGET_PORT | targport | g | ✘ | ✔ | Aleatória (40000 - 60000) | Porta do vítima |
| ARG_LEVEL | level | l | ✘ | ✔ | 1 | Nível do ataque |
| ARG_DURATION | duration | d | ✘ | ✔ | Infinito | timer em segundos |
| ARG_FLOOD | flood | f | ✘ | N/A | Desativado | Ativa modo flood |
| ARG_INCREMENT | inc | i | ✘ | ✔ | Desativado | Ativa modo incremental |
| ARG_AGGRESSIVE | aggressive | a | ✘ | N/A | Desativado | Ativa modo agressivo |
| ARG_CONFIG | config | c | ✘ | ✘ | N/A | Arquivo de configuração |
| ARG_DNS_DOMAIN | domain-name | D | ✘ | ✔ | ddos.dns.com | Parâmetro específico do mirror DNS |
| ARG_SSDP_UPNP_VERSION | upnp-version | V | ✘ | ✔ | UPNP_V1_0 | Parâmetro específico do mirror SSDP |
| ARG_SSDP_UNICAST | unicast | U | ✘ | N/A | N/A | Parâmetro específico do mirror SSDP |
| ARG_SNMP_COMMUNITY_STRING | comunity-string | C | ✘ | ✔ | public | Parâmetro específico do mirror SNMP |
| ARG_SNMP_MAX_REPETITIONS | max-repetitions | R | ✘ | ✔ | 2000 | Parâmetro específico do mirror SNMP |
| ARG_COAP_SZX | szx | Z | ✘ | ✔ | 6 | parâmetro específico do mirror COAP |
| ARG_COAP_URI_PATH | uri-path | P | ✘ | ✔ | .well-known/core | Parâmetro específico do mirror COAP |

As opções e argumentos aceitos também podem ser visualizados em linha de comando pela execução de:

```
lhf --help
```

# Arquitetura

Existem 4 módulos base na implementação do Linderhof:

1. Interface
2. Commander
3. Hall of Mirrors
4. Injector

## Interface

É usada para acessar o Linderhof na máquina local. Divida em duas partes: Interface e Logger. Na Interface é montado um rascunho (draft) do ataque que será executado pelo restante da implementação. É aqui que são setadas as informações passados por linha de comando como o IP do alvo, o IP do amplificador e etc. O Logger regista as informações de saída do Injetor.

## Commander

É composto por três partes principais: o Commander, o Manager e o Hall of Mirros.

O Commander é responsável por validar o draft e inicializar a ferramenta de ataque (seta os handlers de sinal e funções de erro).

O Manager é por fazer a chamada do ataque de acordo com o mirror desejado. Ele aloca na estrutura do draft a função de chamada do mirror e ativa o hall of mirros.

## Hall of Mirrors (hom)

O Hall of Mirror contém todos os protocolos que podem ser explorados no ataque. Aqui é o forjado o pacote que será passado para o injetor. Tal pacote difere de acordo como o protocolo escolhido.

## Injector

É o injetor de pacotes. Ele é composto por duas partes: o Controler e o Injetor. Dentro do módulo Injector existe uma estrutura injector que consiste na thread e outras informações de operação para injeção dos pacotes. Assim, ao longo dessa documentação Injector (com letra maiúscula) diz respeito ao módulo e injector (com letra minuscula) diz respeito a estrutura.

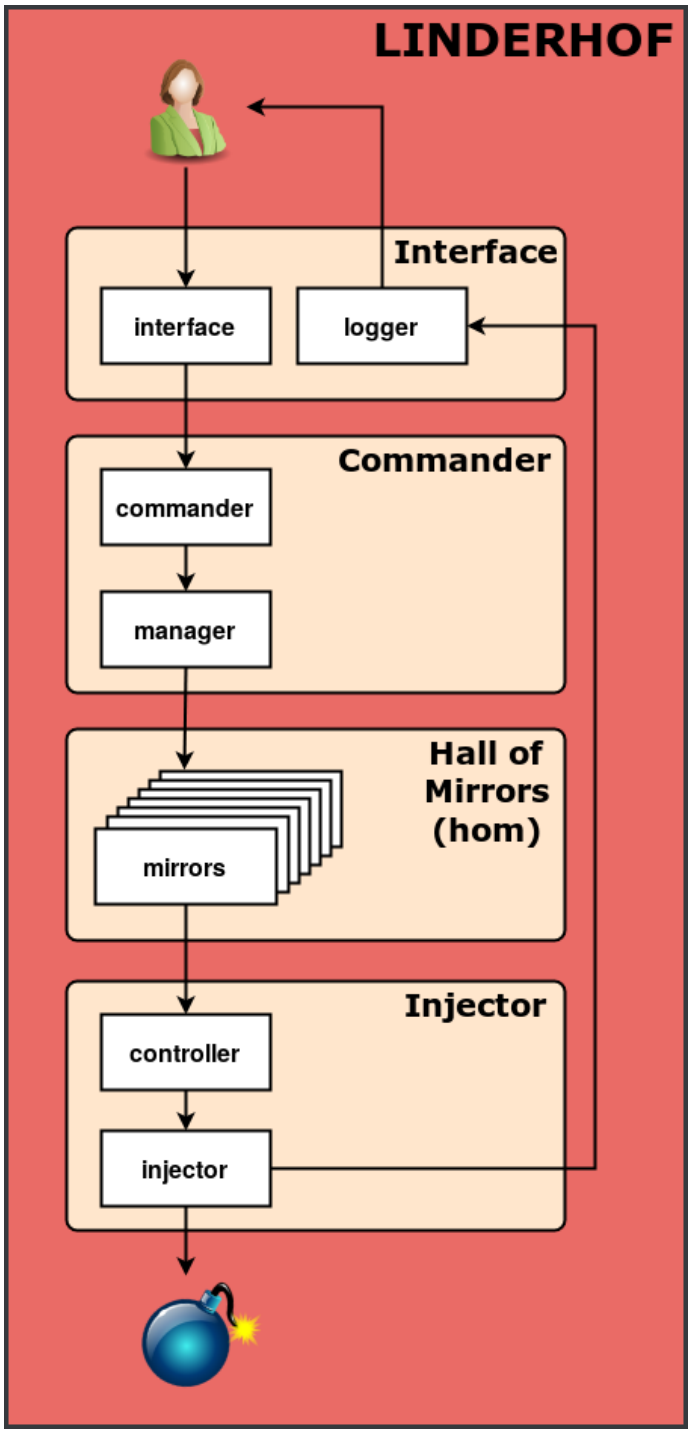


Figure 1: Arquitetura do Lhf.

O Injetor cria e destrói as theads injetoras. Esse processo é controlado por um bucket que corresponde à quantidade de pacotes que ele deve enviar. O controller define a taxa de injeção do ataque controlando esse bucket. Cada thead injetora tem um handler responsável pelo envio dos pacotes.

# Modos de Ataque

Existem quatro modos de intensidade do ataque no Linderhof. Três desse modos são ativados por linha de comando.

O nível do ataque é o controle de intensidade. A intensidade do ataque corresponde à quantidade de pacotes que o Injector deve enviar por segundo. Essa quantidade respeita a fórmula onde L corresponde à intensidade desejada:

```
10^(L-1)
```

A relação nível por intesidade é exemplificada na tabela abaixo. É importante ressaltar que o nível indica a quantidade de pacotes desejada no ataque. Isso não significa que a máquina atacante conseguirá gerar a quantidade de pacotes desejada e nem que a quantidade de pacotes gerados chegará ao refletor.

| Nível | Pacotes/Segundo |
|---|---|
| 1 | 1 |
| 2 | 10 |
| 3 | 100 |
| 4 | 1000 |
| 5 | 10000 |
| 6 | 100000 |
| 7 | 1000000 |
| 8 | 10000000 |
| 9 | 100000000 |
| 10 | 1000000000 |

## Modo Default

Na execusão default do Linderhof, são gerados X pacotes por nível (de acordo com a tabela a cima) e esses pacotes são distribuídos igualmente entre cada reflector. Assim, em uma execução no nível 3 com 4 refletores, cada thread injetora terá um bucket de 25 pacotes. Ou seja, desejamos que cada refletor receba 25 pacotes.

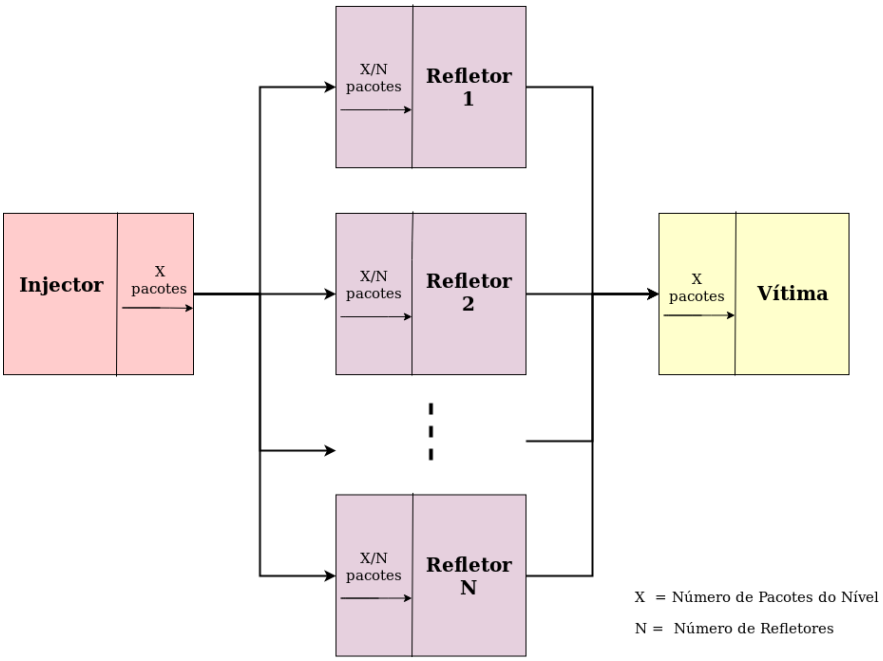


Figure 2: Modo Default.

## Modo Incremental (parâmetro -i)

No modo incremental, a ferramenta aumenta o nível do ataque de acordo com a frequência passada pelo usuário. Assim, se o parâmetro recebido for -i 2, a ferramenta irá mandar 1 pacote pelos 2 primeiros segundos, 10 pacotes nos 2 segundos seguintes, 100 pacotes nos próximos 2 segundos e assim por diante.

## Modo Agressivo (parâmetro -a)

O modo agressivo extrapola o nível para os refletores. Ou seja, cada thread injetora receberá a quantidade total de pacotes desejada para aquele nível. Assim, em uma execução no nível 3 com 4 refletores em modo agressivo, cada thread injetora terá um bucket de 100 pacotes e o desejo é que cada refletor receba 100 pacotes.

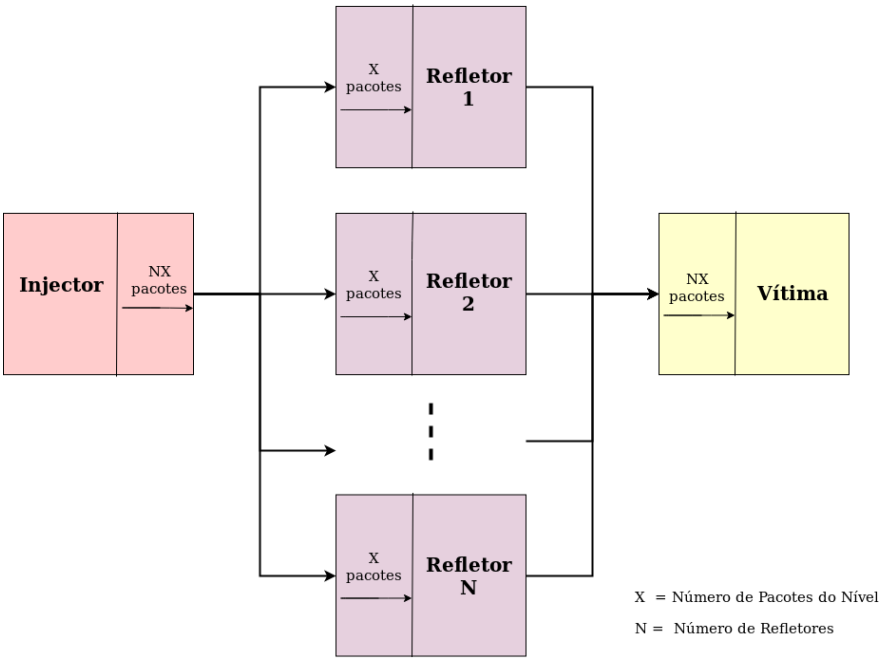


Figure 3: Modo Agressivo.

## Modo Flood (parâmetro -f)

No modo flood não existem níveis. Ou seja, não existe um tamanho para o bucket. Não existe um máximo de pacotes enviados. A ferramenta entra em loop e envia o máximo possível de pacotes para todos os refletores.

# Como os Pacotes são Forjados

A estrutura Packet contém o pacote a ser envidado ao longo da ferramenta, além de algumas outras informações necessárias para a injeção. Assim, por abstração, essa estrutura representa o pacote que será enviado. Esse pacote é montado a partir dos parâmetros passados por linha de comando. A montagem do pacote em si ocorre no módulo Hall of Mirrors.

Antes dessa montagem as informações do pacote são passada pelos módulos com ajuda da estrutura draft contida em outras estruturas auxiliares.

A figura abaixo exemplifica essas estruturas. As variáveis em vermelho são as do tipo LhfDraft.

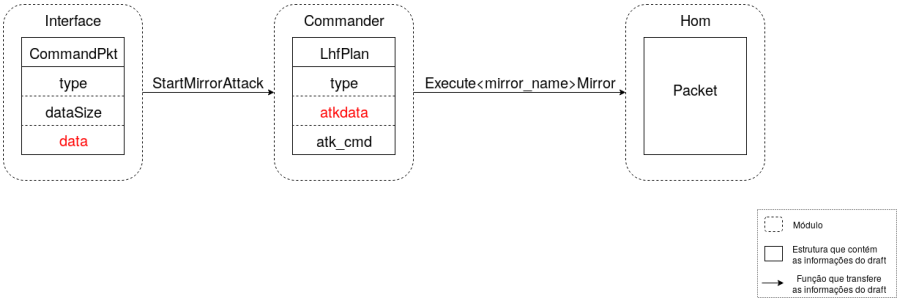


Figure 4: Estruturas auxiliares na transmissão do draft.

Nem todas as informações do draft são diretamente passadas para o pacote. A figura abaixo representa quais informações do draft são mapeada para a estrutura final do pacote.

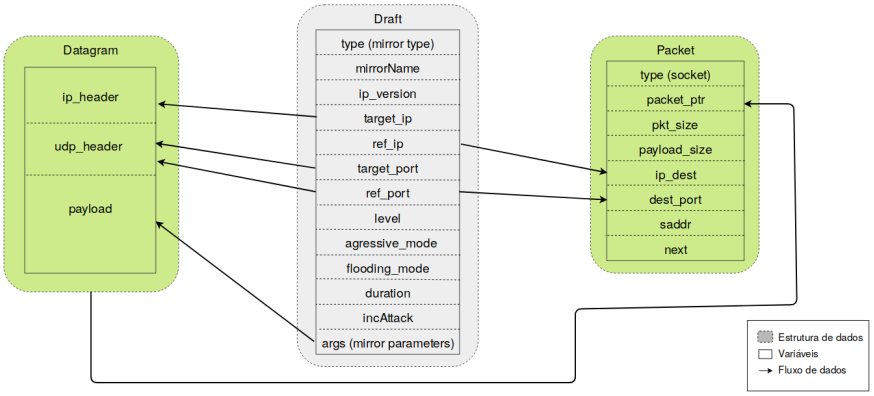


Figure 5: Draft X Packet.

No módulo Hall of Mirror, as informações do draft e as informações necessárias referentes ao protocolo (mirror) escolhido são repassada para o pacote.

A figura abaixo mostra o caminho dos dados passados pelo usuário até sua inclusão no pacote.

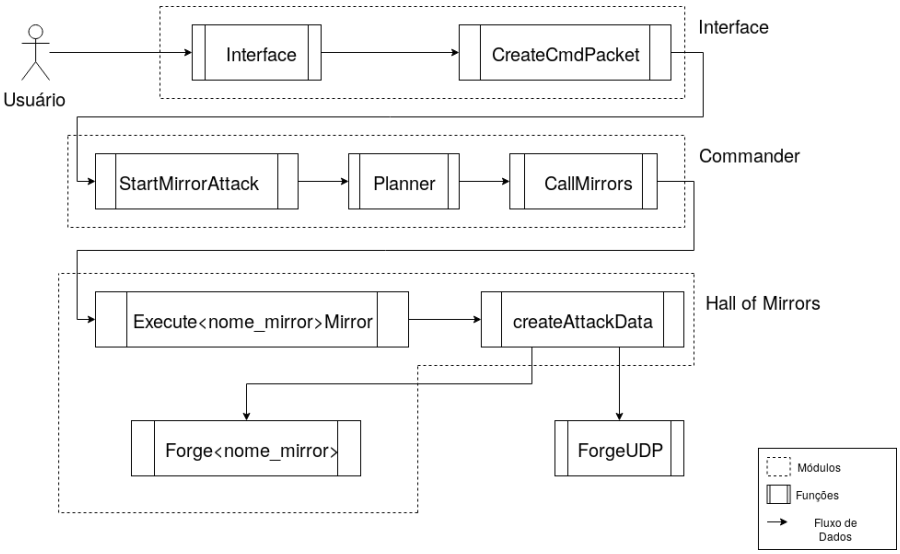


Figure 6: Caminho dos dados de entrada.

## Como acontece a injeção de pacotes

O módulo Injector é chamado pelo módulo Hall of Mirrors. O primeiro passo é criar as threads injetoras e seus buckets. O bucket corresponde a quantidade de pacotes que devem ser enviados. O controller é responsável por controlar a taxa de injeção, controlando esse bucket.

Cada refletor representa uma thread injetora. Assim, o número de injectors envolvidos no ataque representa o número de refletores informados.

Atualmente a ferramenta utiliza apenas um pacote que é enviado repetidas vezes para o refletor. Caso o ataque seja incremental, a porta de origem do pacote, junto com o nível, é incrementada para melhorar a análise. Essas mudanças ocorrem no controler.

Como apenas um pacote é enviado, a estrutura do bucket é na verdade uma abstração. A implementação real é representada apenas pelo número de pacotes que devem ser enviados. Ou seja, quando um pacote é enviado a variável bucket é decrescida. Esse controle de bucket envolve três variáveis: bucketMax, que

representa o tamanho máximo do bucket, ou seja, quantos pacotes devem ser enviados no máximo no nível atual pelo injetor; bucketSize, que representa o tamanho atual do nível; freeBucket que é uma flag e representa o fim do ataque.

A figura abaixo representa todos os passos da injeção de pacotes.

Todos os modos de ataque fazem uso do mesmo mecanismo de injeção.

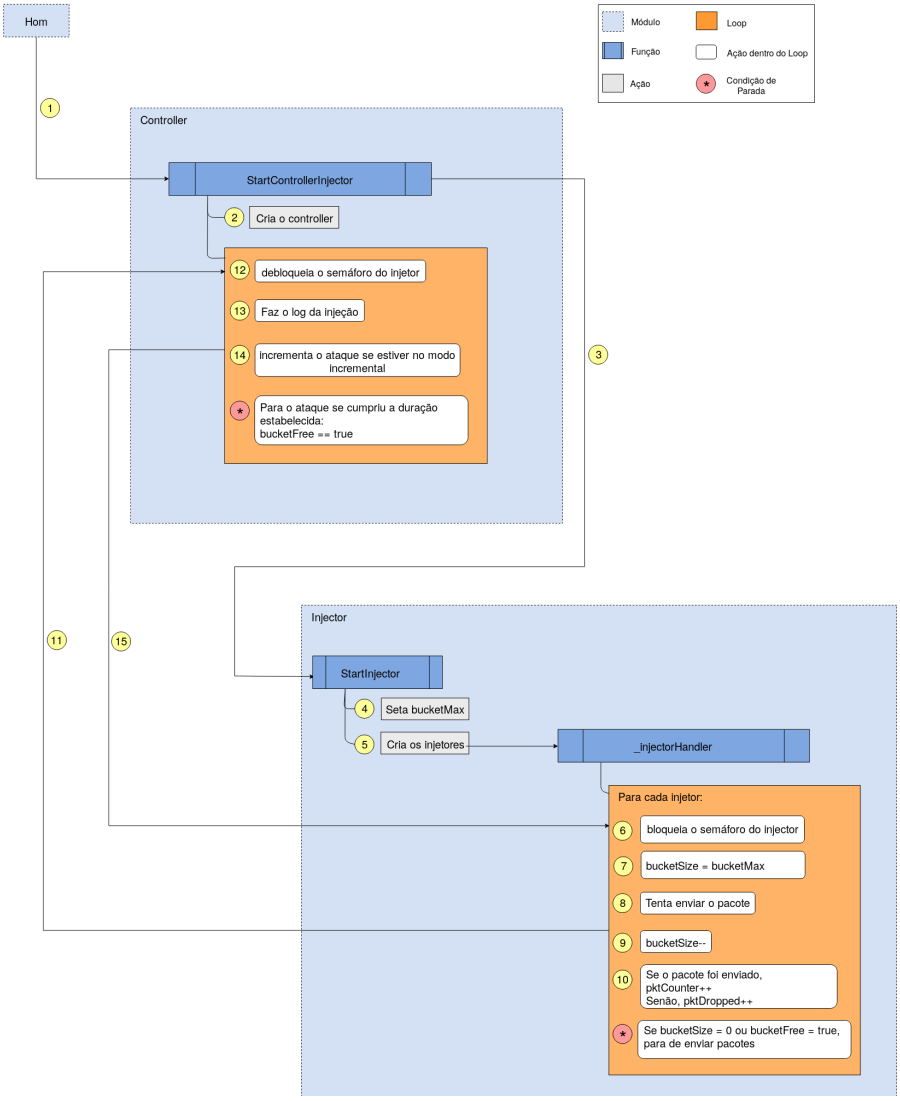


Figure 7: Processo de Injeção de Pacotes.