TÉCNICO LISBOA

SEGURANÇA DE SOFTWARE

Mestrado Integrado em Eng. Informática e de Computadores

RELATÓRIO

Grupo 38

NICOLAS LEE GUIDOTTI - 92242

JOSUÉ HENRIQUE DE FREITAS ANDRADE - 92185

SUMÁRIO

DESIGN

O programa de detecção de vulnerabilidades foi desenvolvido utilizando técnicas de design orientado a objetos, de forma a deixar o sistema mais modular e testável, ele possui 6 arquivos sendo o propósito de cada um introduzido abaixo:

CodeAnalyzer.hpp: É o arquivo header de '*CodeAnalyzer.cpp'*, possui a declaração da classe '*CodeAnalyzer'* e das diversas funções utilizadas durante a leitura, escrita e busca de vulnerabilidades.

CodeAnalyzer.cpp: Neste arquivo está presente às implementações da leitura e da escrita em um arquivo *JSON*, além da implementação do código que realizam efetivamente a análise das vulnerabilidades e indicam a existência delas.

Registers.hpp: Neste arquivo ocorre a declaração da classe *'Registers'* essencial para o funcionamento, pois ela é utilizada para simular o funcionamento dos registradores durante a análise do código.

Registers.cpp: Possui a implementação de todas as funções da classe *'Registers'* que são utilizadas na manipulação dos registradores e na busca de vulnerabilidades.

StructDefinition.cpp: Este arquivo declara as estruturas que são utilizadas para a busca e armazenamento de vulnerabilidades.

main.cpp: Esta é a função principal que inicia a execução do programa.

PRINCIPAIS DECISÕES DE DESIGN

- ***StructDefinition***

Uma das principais decisões de design é definição das estruturas que serão necessárias para analisar o código. Devido a isso, desenvolvemos 5 estruturas *(Variable, Instruction, Function, Vulnerability, MemoryStack)* que juntas com a classe *Registers* manipulam todos os valores desde a leitura do input até a escrita das vulnerabilidades no output.

A *struct Variable* é responsável por armazenar as variáveis e seus respectivos valores obtidos pela função *readJSON,* e além disso ela mantém informações que serão utilizadas a posteriori como o tamanho efetivo *(effective_size)* e uma ponteiro para outras variáveis *(merge_var)* que será usado caso durante a execução do código alguma variável tente se fundir com outra variável.

A *struct Instruction* armazena as instruções lidas do arquivo *JSON*, sendo que os argumentos são armazenados usando a objetos da classe *std::map o que* facilita o acesso aos elementos durante a execução.

A *struct Function* possui um vetor para as variáveis e outro para as intruções de forma a armazenar todos os valores do arquivo *JSON*, além disso, possui um inteiro *(current_inst)* que ajuda no gerenciamento da execução das instruções.

A *struct MemoryStack* é de crucial importância para o projeto, pois ela é responsável pelo simulação do stack de memória, ela possui internamente dois objetos da classe *std::map* um deles é responsável por armazenar um endereço de memoria e uma variáveis *(var)* e o outro armazena um endereço de memória e um registrador *(const_value).*

- ***Registers***

Na classe *Registers*, as principais decisões foram feitas de forma a deixar a classe simular corretamente um registrador, para isso ela conta com 2 elementos da classe *std::map*. O primeiro é o *reg_var,* ele foi pensado de forma a guardar o nome do registrador e um ponteiro para o endereço de uma variável, e o outro é o *reg_const* que guarda o nome do registrador e um valor inteiro correspondente ao endereço.

Além disso, ela ainda tem 2 métodos para cada um dos elementos apresentados, o primeiro método *(getVarRegister ou getConstRegister)* tem como retorno uma tuple que indica a existência do registrador, e o segundo método *(addRegister)* é utilizado na adição de novos registradores.

- ***CodeAnalyzer***

A classe *CodeAnalyzer* é responsável pela a execução de toda a lógica utilizada na busca de vulnerabilidades, e ela possui diversos pontos que são considerados cruciais para o funcionamento da aplicação. Para o funcionamento correto, ela possui 4 elementos, que são, uma variável do tipo *MemoryStack (mem_stack),* um registrador da classe *Registers (reg),* um objeto da classe *std::map (functions),* que é mapeado utilizando o nome da função e uma objeto da classe *Function,* e por fim, um elemento da classe *std::vector (vulnerabilities)* utilizado para listar todas as vulnerabilidades encontradas.

Para a leitura do arquivo *JSON* que possui todos os dados do código em assembly, utiliza-se as funções *readJSON* e *jsonToStruct*, esses métodos foram implementados de forma a realizarem a leitura do arquivo e o carregamento da variável *functions.*

A análise do código é feita através do encadeamento dos métodos da família *analyze*, de modo que o escopo da verificação se torna cada vez mais específica conforme percorre-se cada etapa. Com isso em mente, a análise começa com o método *analyze*, que é responsável por analisar o código em nível global, ordenando as chamadas de funções em uma pilha. A primeira função colocada na pilha é aquela denominada como *main*. Se a pilha não estiver vazia, chama o método *analyzeFunction*, passando como parâmetros a função que vai ser analisada (ou seja, aquela no topo da pilha) e a própria pilha.

O método *analyzeFunction* é responsável, como o nome já diz, por analisar uma função do código. A análise é feita de forma similar a execução, ou seja, o método *analyzeFunction* percorre as instruções da função e efetua as operações

correspondentes. Mas diferente da execução, as operações podem ser tanto com números como com as variáveis da função. Além disso, considerou-se um modelo simplificado com apenas as seguintes instruções: *ret*, *leave*, *nop*, *push*, *call*, *mov*, *lea*, *sub* e *add*. Dentre essas instruções, há alguns pontos importantes. As instruções *add* e *sub* operam apenas sobre os *rbp* (*base pointer*) e o *rsp* (*stack pointer*). A instrução *leave* chama o método *desallocFunction* que remove o espaço ocupado pela função e restaura os valores de *rbp*, *rsp* e *rsi*, bem como remove a função da pilha. A instrução *ret* retorna para o método *analyze*. A instrução *call* verifica se a função chamada está no código, e se tiver, aloca o espaço necessário na memória e coloca os valores adequados nos registradores *rbp*, *rsp* e *rsi* através do método *allocFunction*. Caso o contrário, chama o método *analyzeCalledFunction* passando para a próxima etapa da análise.

O método *analyzeCalledFunction* verifica, então, se a função chamada é uma função vulnerável (*gets*, *strcpy*, *strcat*, *fgets*, *strncpy*, *strncat*, *sprintf*, *scanf*, *fscanf*, *snprintf* ou *read*). Se sim, recupera os argumentos da função através dos registradores (os argumentos seguem uma ordem pré-definida de registradores), calcula a quantidade de bytes utilizado pelo *buffer* resultante (todas as funções vulneráveis consideradas realizam operações com *buffers*) e chama a função *analyzeOverflow*. O overflow é calculado através da diferença entre o tamanho efetivo e real do *buffer* resultante. Cabe ressaltar que a função *strncpy(buffer1, buffer2, num)* não coloca /0 no final do *buffer1* se o tamanho do *buffer2* for maior que *num*, e por isso, pode haver concatenação de *buffers*. Além disso, considerou-se que o formato da entrada das funções *scanf*, *fscanf* e *sprintf* apenas pode ser “%s” ou “%s %s”. Para descobrir a quantidade de *buffers* envolvidos, verificou-se a quantidade de registradores que estão ocupados. Por exemplo, a função *scanf* considerada pode ter até 3 argumentos (o formato, o *buffer1* e o *buffer2*), então, se houver apenas uma variável no registrador *rsi*, existe apenas o *buffer1*. Se existir variáveis no registrador *rsi* e no registrador *rdx*, existe tanto o *buffer1* como o *buffer2*. Esse modelo apresenta como principal problema o fato de que uma variável armazenada em um registrador não significa que ela será utilizada como argumento, de modo que o formato “%s” possa ser interpretado como “%s%s”. Uma forma de corrigir isso, seria identificar a *string* que caracteriza o formato da entrada durante a análise das instruções da função, e armazená-la até a chamada da função *scanf*, *fscanf* ou *sprintf*. De forma, poderia então analisar a string e assim determinar a quantidade de argumentos, bem como os seus tipos.

O último método é o *analyzeOverflow*, que é responsável por determinar se houve *stack overflow*, e em caso positivo, determinar quais as vulnerabilidades associadas com esse *overflow*, armazenando-as no vetor *vulnerabilities*.

A escrita no arquivo de *JSON* no formato especificado é feita pelos métodos *writeJSON* e *structToJson* que retiram as vulnerabilidades descobertas na vetor *vulnerabilities* e escreve seus valores em um arquivo com o final *“.output.json”.*

RESULTADOS OBTIDOS

O programa apresentou resultados similares ao disponibilizados nos arquivos de teste. Algumas divergências, inicialmente, foram encontradas mas os arquivos foram desconsiderados como sugerido pelos professores no *slack.*

Abaixo se encontra um exemplo de entrada e saída do programa, é possível notar que o programa funcionou corretamente nesse exemplo, calculando inclusive as vulnerabilidades após a chamada da função *func1.*

Os testes foram satisfatórios para todos os arquivos disponibilizados e para alguns criados, apresentando sempre o resultado igual ao dos arquivos disponibilizados na página ou com poucas diferenças devido a detecção de vulnerabilidades mais complexas.

Input: *23_fgets_fun_main_nok.json*	**Output:** *23_fgets_fun_main_nok.output.json*
{ "main": { "Ninstructions": 15, "variables": [{ "bytes": 4, "type": "int", "name": "control", "address": "rbp-0x4" }, { "bytes": 64, "type": "buffer", "name": "buf", "address": "rbp-0x50" }], "instructions": [{ "op": "push", "pos": 0, "args": { "value": "rbp" }, "address": "400589" }, { "op": "mov", "pos": 1, "args": { "dest": "rbp", "value": "rsp" }, "address": "40058a" }, { "op": "sub", "pos": 2, "args": { "dest": "rsp", "value": "0x50" }, "address": "40058d" }, { "op": "mov", "pos": 3, "args": { "dest": "DWORD PTR [rbp-0x4]", "value": "0x17" }, "address": "400591" }, { "op": "mov", "pos": 4, "args": { "dest": "rdx", "obs": "# 601040 <stdin@@GLIBC_2.2.5>", "value": "QWORD PTR [rip+0x200aa1]" }, "address": "400598" }, { "op": "lea", "pos": 5, "args": { "dest": "rax", "value": "[rbp-0x50]" }, "address": "40059f" }, { "op": "mov", "pos": 6, "args": { "dest": "esi", "value": "0x64" }, "address": "4005a3" }, { "op": "mov", "pos": 7, "args": { "dest": "rdi", "value": "rax" }, "address": "4005a8" }, { "op": "call", "pos": 8, "args": { "fnname": "<fgets@plt>", "address": "400480" }, "address": "4005ab" }, { "op": "lea", "pos": 9, "args": { "dest": "rax", "value": "[rbp-0x50]" }, "address": "4005b0" }, { "op": "mov", "pos": 10, "args": { "dest": "rdi", "value": "rax" }, "address": "4005b4" }, { "op": "call", "pos": 11, "args": { "fnname": "<fun1>", "address":	[{ "address": "4005ab", "fnname": "fgets", "overflow_var": "buf", "overflown_addr": "rbp-0x10", "vuln_function": "main", "vulnerability": "INVALIDACC" }, { "address": "4005ab", "fnname": "fgets", "overflow_var": "buf", "overflown_var": "control", "vuln_function": "main", "vulnerability": "VAROVERFLOW" }, { "address": "4005ab", "fnname": "fgets", "overflow_var": "buf", "vuln_function": "main", "vulnerability": "RBPOVERFLOW" }, { "address": "4005ab", "fnname": "fgets", "overflow_var": "buf",

"400567" }, "address": "4005b7" }, { "op": "mov", "pos": 12, "args": { "dest": "eax", "value": "0x0" }, "address": "4005bc" }, { "op": "leave", "pos": 13, "address": "4005c1" }, { "op": "ret", "pos": 14, "address": "4005c2" }] }, "fun1": { "Ninstructions": 12, "variables": [{ "bytes": 16, "type": "buffer", "name": "buf2", "address": "rbp-0x10" }], "instructions": [{ "op": "push", "pos": 0, "args": { "value": "rbp" }, "address": "400567" }, { "op": "mov", "pos": 1, "args": { "dest": "rbp", "value": "rsp" }, "address": "400568" }, { "op": "sub", "pos": 2, "args": { "dest": "rsp", "value": "0x20" }, "address": "40056b" }, { "op": "mov", "pos": 3, "args": { "dest": "QWORD PTR [rbp-0x18]", "value": "rdi" }, "address": "40056f" }, { "op": "mov", "pos": 4, "args": { "dest": "rdx", "value": "QWORD PTR [rbp-0x18]" }, "address": "400573" }, { "op": "lea", "pos": 5, "args": { "dest": "rax", "value": "[rbp-0x10]" }, "address": "400577" }, { "op": "mov", "pos": 6, "args": { "dest": "rsi", "value": "rdx" }, "address": "40057b" }, { "op": "mov", "pos": 7, "args": { "dest": "rdi", "value": "rax" }, "address": "40057e" }, { "op": "call", "pos": 8, "args": { "fnname": "<strcpy@plt>", "address": "400470" }, "address": "400581" }, { "op": "nop", "pos": 9, "address": "400586" }, { "op": "leave", "pos": 10, "address": "400587" }, { "op": "ret", "pos": 11, "address": "400588" }] } }

```
    "vuln_function": "main",
    "vulnerability": "RETOVERFLOW"
  },
  {
    "address": "4005ab",
    "fnname": "fgets",
    "overflow_var": "buf",
    "overflown_addr": "rbp+0x10",
    "vuln_function": "main",
    "vulnerability": "SCORRUPTION"
  },
  {
    "address": "400581",
    "fnname": "strcpy",
    "overflow_var": "buf2",
    "vuln_function": "fun1",
    "vulnerability": "RBPOVERFLOW"
  },
  {
    "address": "400581",
    "fnname": "strcpy",
    "overflow_var": "buf2",
    "vuln_function": "fun1",
    "vulnerability": "RETOVERFLOW"
  },
  {
    "address": "400581",
    "fnname": "strcpy",
    "overflow_var": "buf2",
    "overflown_addr": "rbp+0x10",
    "vuln_function": "fun1",
    "vulnerability": "SCORRUPTION"
  }
]
```

ANÁLISE E DISCUSSÃO

METAS ATINGIDAS

A nossa ferramenta é capaz de identificar overflow em todas as funções especificadas (tanto para a análise básica como avançada). Além disso, o nosso programa identifica todas as instruções básicas especificadas (e também o *call* de funções genéricas), de modo que consiga acompanhar o fluxo de informações do programa. Por fim, também é capaz de ler e escrever um arquivo JSON. Dessa forma, o nosso programa é capaz de encontrar todas as vulnerabilidades dos exemplos disponibilizados (exceto os 3 últimos que envolvem o acesso direto à memória).

LIMITAÇÕES e EXTENSÕES

Embora nosso programa consiga atingir os objetivos propostos, ainda possui algumas limitações. Uma delas é o fato do formato da entrada poder ser interpretado equivocadamente (considerando “%s%s” ao invés de “%s”) nas funções *scanf*, *fscanf* e *snprintf*. Outro ponto importante é que o nosso modelo para essas funções é muito simples, enquanto essas funções na realidade suportam um número quase infinito de formatos de entrada. Esse limitação e solução já foram mencionadas anteriormente.

Além disso, uma grande melhoria na segurança seria verificar se a constante colocada na memória está indexada a uma variável ou não, de forma a evitar e detectar um acesso inválido.

Uma outra limitação é fato do nosso programa não considerar desvios condicionais (*je*, *jne*, *cmp*, *test*) ou incondicionais (*jmp*).

Por fim, como a nossa análise é estática e simplificada, detecta apenas vulnerabilidades referentes ao *stack overflow*. Outras tipos de vulnerabilidades como *race conditions* e *invalid format strings* não são reconhecidos.

As principais extensões que podem ser desenvolvidas para melhorar a segurança são referentes a solução das limitações demonstradas acima. Uma possível extensão seria a adição de outras funções, como por exemplo *je, jmp, jne*, *cmp, test,* entre outras, e a adaptação das já existentes para entradas com formato diferente para que o programa funcione para uma quantidade maior e mais diversa e código similar à realidade.

A detecção de outros tipos de vulnerabilidades pode ser feito ao implementar um modelo mais genérico, bem como efetuando a análise dinâmica do código.