ВИСШЕ ВОЕННОМОРСКО УЧИЛИЩЕ „Н. Й. ВАПЦАРОВ“ ФАКУЛТЕТ „ИНЖЕНЕРЕН“ - КАТЕДРА „ИНФОРМАЦИОННИ ТЕХНОЛОГИИ“

КУРСОВ ПРОЕКТ

на тема „ДЕКОМПИЛАЦИЯ И АНАЛИЗ НА ИЗПЪЛНИМИ ФАЙЛОВЕ ЧРЕЗ ГРАФИЧНА РЕПРЕЗЕНТАЦИЯ“

ДИСЦИПЛИНА:

„Курсов проект“

Студент: Антон Евгениев Атанасов

Специалност: Киберсигурност

Фак. № 12621101

Дата: 18.12.2024 г.

гр. Варна

1. Въведение

Изпълнимите файлове, известни още като двоични файлове, играят основна роля в съвременната компютърна архитектура, като съдържат машинния код, който процесорът може да интерпретира и изпълнява. Те представляват резултат от процеса на компилация, при който програмен код, написан на високо ниво, се превръща в низ от инструкции, разбираеми за конкретния хардуер. Тези файлове обикновено са резултат от комбинирането на различни обектни файлове и библиотеки, чрез линкване, което ги прави готови за директно изпълнение.

Изпълнимите файлове са оптимизирани да бъдат ефективни за компютърната машина, но трудни за разбиране от хора. Те съдържат двоични данни, които са нечетими за човешкото око и които изискват специализирани инструменти, като дисасемблери и дебъгери, за да бъдат интерпретирани. Тази характеристика прави двоични файлове ефективни за обработка от предназначен хардуер, но предизвиква трудности при анализ и извършване на ръчни модификации, което е важен аспект при проучването на сигурността на изпълними програми и анализирането на техния вътрешен механизъм.

Работата с двоични файлове е от съществено значение за тестването на софтуера и отстраняването на грешки, тъй като често те съдържат финалната версия на програма, която трябва да бъде тествана в реална среда. Двоичният формат позволява директно изпълнение на програмата, което е необходимо за откриване на бъгове и несъответствия в поведението на софтуера, особено върху разновидности от хардуер или работата с драйвери и фърмуер. Освен това, при работата с такива файлове могат да бъдат открити проблеми, които не се проявяват по време на компилирането или тестовете на изходния код. Дисасемблирането и анализирането на двоичните файлове могат да помогнат за локализирането на проблеми, които не са видими на ниво изходен код.

Документирането на проекта също е важна цел при работата с двоични файлове. Въпреки че двоичният код не е предназначен да бъде четим от хора,

анализирането му може да предостави ценна информация за структурата и функционалността на софтуера, която може да бъде включена в документацията на проекта и да посочи нуждата от оптимизация на изчисленията, тяхната комплексност и ефикасността при изпълнение върху целевата за него архитектура. Тези анализи често разкриват специфични алгоритми, методи и важни зависимости, които не са винаги очевидни в изходния код. Чрез обратен инженеринг на двоичния файл, разработчиците могат да създадат подробни описания на програмата, които са важни за бъдещи модификации или за поддръжка на софтуера.

Проверка на безопасността на кода също е критичен аспект за киберсигурността, при който анализата на двоични файлове играе важна роля. Двоичните файлове могат да съдържат уязвимости, които не са видими в изходния код и могат да бъдат експлоатирани от зловреден софтуер. Прегледът на двоичния код чрез специализирани инструменти позволява откриването на тези уязвимости, които могат да водят до сериозни последствия за сигурността на системите. Това включва анализ на буферни препълвания, неконтролирани променливи и незащитени сесии и нишки на процеси, които биха могли да застрашат защитата на данни или да доведат до компрометиране на системата.

В контекста на анализа на малуер, работата с двоични файлове е от особено значение. Малуерът обикновено се представя под формата на двуичен код, който е предназначен да избягва лесното откритие и декомпилиране. Чрез извършване на обратен инженеринг върху двоичния файл на малуера, изследователите могат да идентифицират неговата структура, да разберат начините му на действие и да открият как точно засяга системите, в които е инсталиран. Това е ключов процес за създаване на антивирусни решения и за осигуряване на защитата на компютърните мрежи.

Образователните цели също играят важна роля при анализа на двоични файлове. Чрез изучаването на структурата на изпълнимите файлове и техния начин на работа, студентите и специалистите по сигурността могат да разберат

дълбочината на компютърните науки и механиката на програмите. Това знание е основа за по-добро разбиране на процесите на компилация, линкване и оптимизация, които формират основата на всяко компютърно приложение.

Липсата на отворен код е обикновено е главната причина, поради която работата с двоични файлове е необходима в неспециализирани хардуерни случаи. В ситуации, в които програмите не разполагат с изходен код или той не е публично достъпен, единственият начин да се разбере как функционира даден софтуер е чрез анализ на двоичния му екземпляр. Това може да бъде необходимо за извършване на корекции, оптимизации или за откритие на проблеми в съществуващите програми, които не са били разкрити публично.

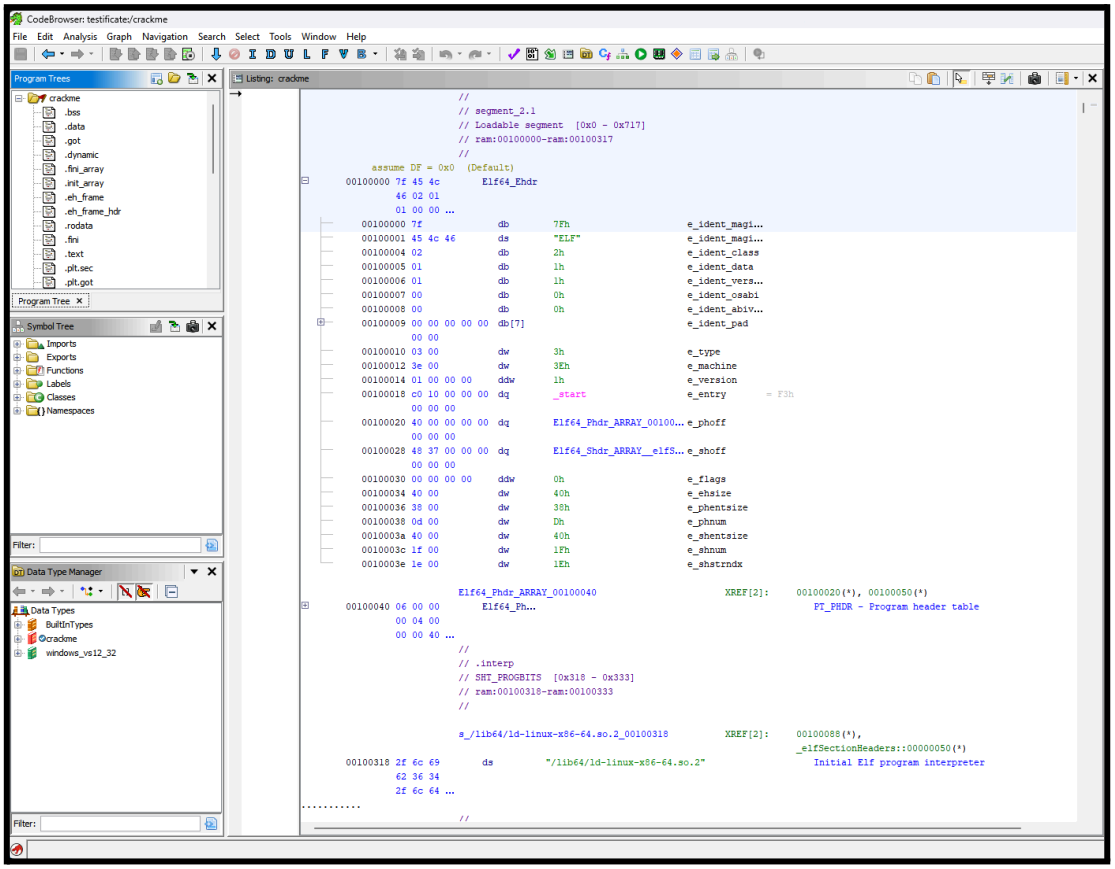
Основната нужда проектът, който предстои да бъде описан се дължи на недостигът от специалисти, които са обучени да работят с декомпилираната форма на код, която често е представена в асемблерен език, близък до машинния код и трудни за разбиране от хора. Тези специалисти трябва да разбират подробно архитектурата на процесора и вътрешните механизми на програмата. Поради сложността на асемблера, нараства нуждата от инструменти, които да визуализират логиката на програмата на по-високо ниво, като например графични интерфейси или динамични диаграми. Това позволява бързо разбиране на основните функционалности и алгоритми на софтуера, като същевременно се увеличава ефективността на анализа и се намалява времето за обработка на сложни кодови структури.

2. Функционалности

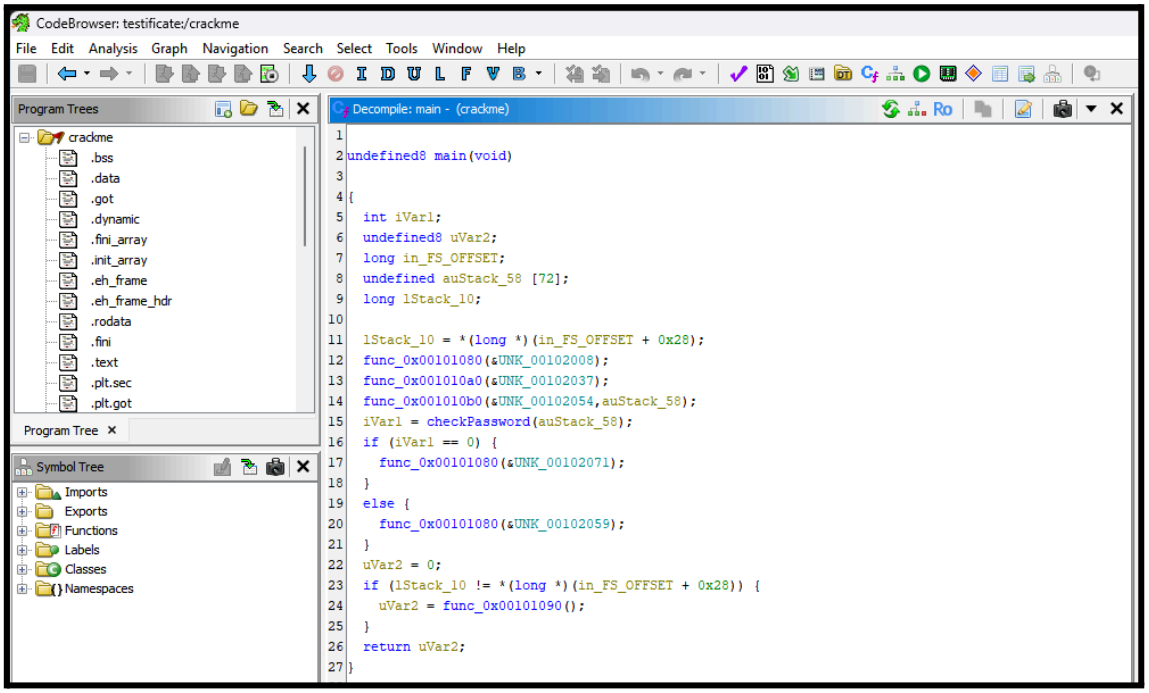
Зареждането на изпълними и двоични файлове се постига с програмен инструмент, наречен декомпилатор, който създава проект за изпълнимия файл, анализира го и съхранява извлечените данни, заедно с потенциалните промени, които потребителя може да нанесе в процеса на обработка на програмата.

След успешното зареждане на проекта, декомпилаторът дава възможност за декомпилация и деасемблиране на кода. Първоначално, програмата генерира

асемблерен код, който представя инструкциите на процесора в разбираем за специалистите вид [Фиг. 1.]. Това включва операции като прехвърляне на данни между регистрите, аритметични изчисления, условни и безусловни преходи. Оттам нататък, инструментът използва вградения си дисасемблер, за да преобразува асемблерния код в код на C-подобен език [Фиг. 2.]. Този процес е изключително полезен, тъй като предоставя по-абстрактна и четима форма на програмата, която позволява на потребителите да проследяват логиката ѝ, да идентифицират алгоритми и да разбират взаимодействието между различните модули. Също така се предоставя и възможност за корекция на декомпилирания код, като потребителят може да задава свои типове данни и коментари, за да улесни по-нататъшния анализ.



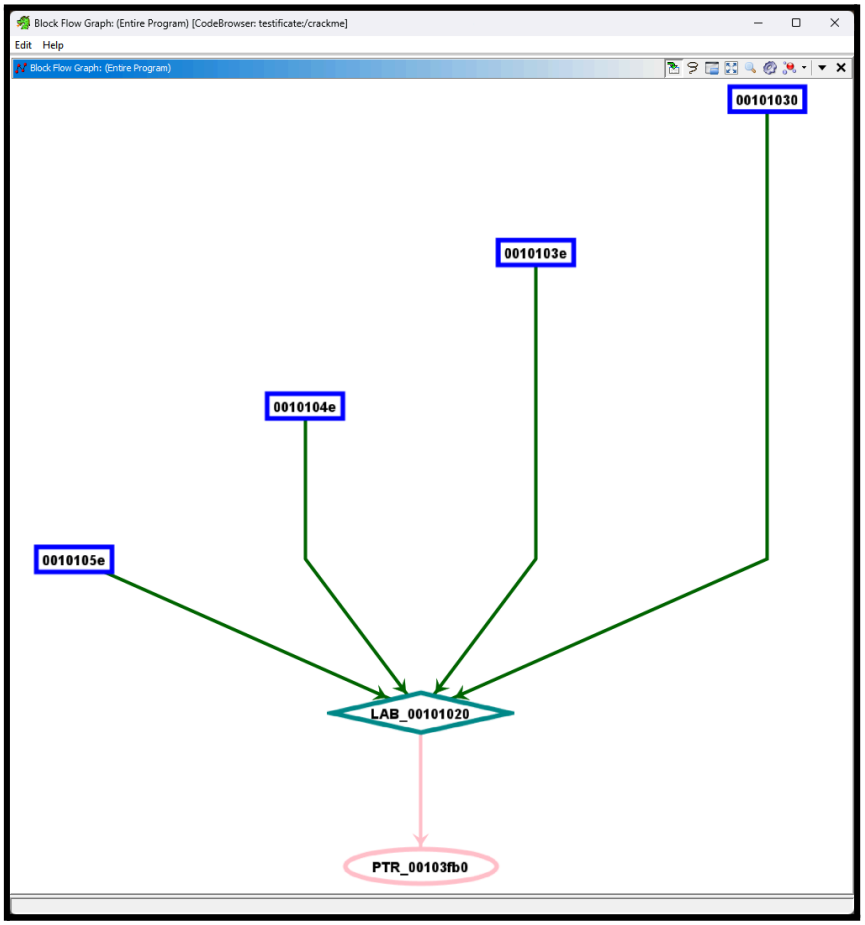
Фиг. 1. Декомпилация на изпълним файл с Ghidra



Фиг. 2. Деасемблиране на изпълним файл с Ghidra

За разлика от програмния код, който е по същество едномерен поток от команди, има структури, които си служат с двумерна репрезентация на логика, с което постигат по-бързо представяне и разбиране на информация. Затова една от най-полезните функции на един декомпилатор е възможността за генериране на диаграми на потока на управление и повикванията на функции [Фиг. 3.]. Тези диаграми показват как различните блокове от асемблерен код са свързани помежду си, както и преходите между тях. Диаграмите могат да включват детайли като условни разклонения, безусловни преходи и взаимовръзки между функциите. Тази визуализация е ключова за разбирането на сложни програми, особено когато те съдържат обфускиран код или многобройни условни операции. Анализаторите могат бързо да идентифицират критични функции, като точки на влизане, системни повиквания или обработка на данни, и да преценят значението им за изпълнението на програмата. Диаграмите също така са незаменими при анализ на малуер, където те могат да покажат поведението на програмата и ключовите функции, отговорни за злонамерените действия. Недостатъкът на резултатът от този анализ е, че блоковата диаграма е строго

ориентирана върху асемблерният код на програмата и е все още трудно четима от неспециализирани потребители.

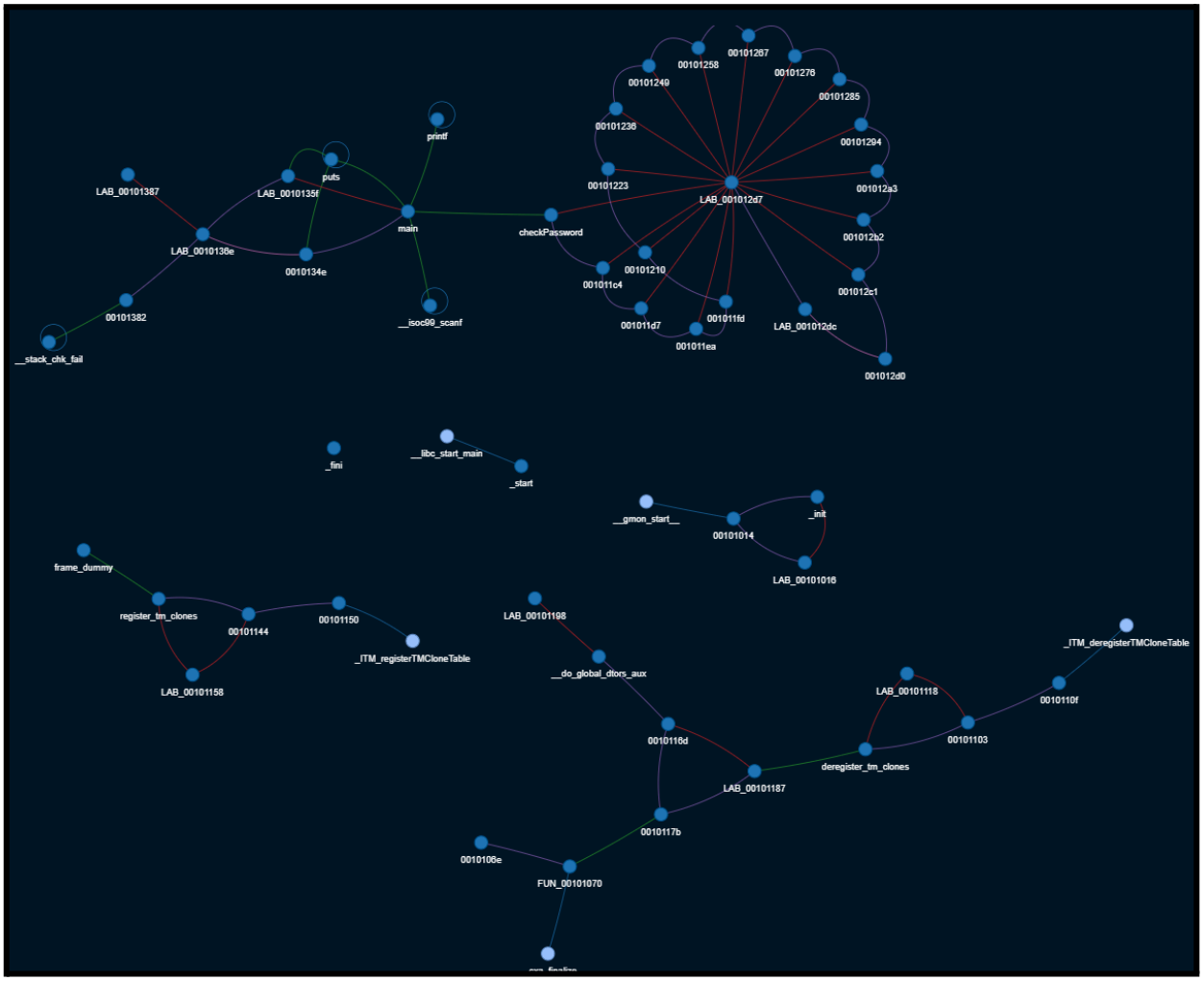


Фиг. 3. Диаграма на повикванията на изпълним файл в Ghidra

За тази цел е имплементирана функционалност, която да създаде динамичен граф от по-високо ниво [1-4], ориентиран към дисасемблираният C-подобен код под формата на модерна карта на знанието (Knowledge Graph), където интуитивно да се представи логиката на анализираната програма. Декомпилаторът предоставя детайлна информация за всеки блок от декомпилирания код, което е особено полезно за по-задълбочени анализи. От всеки блок могат да бъдат извлечени адресите в паметта, където са разположени съответните инструкции и данни. Инструментът автоматично анализира променливите в рамките на блока, включително техните размери, типове и

начини на използване. Анализаторите могат да преглеждат и структурите от данни, които програмата използва, както и връзките между различните функции. Това улеснява идентифицирането на ключови обекти и техните взаимодействия в програмата. Тази информация е особено важна при идентифициране на уязвимости, като неправилно управление на паметта или използване на неподходящи типове данни, което може да доведе до експлоатиране на софтуера.

Чрез извлечената информация за асемблерните блокове се съставя обектно-ориентиран файл, който съдържа същината на деасемблирания код. Чрез функция за динамично изграждане на релациите под формата на стойности в свойствата на обектите се формира диаграма, в която за всеки блок се създава възел с цвят, размер и фигура, които отговарят на определено качество на блока по избор на потребителя. Същата функция образува ръб между възлите, които отговарят на абстрактна релация, отново дефинирана от потребителя - с цвят и тегло, които посочват нейния тип и значимост. За представителен пример най-често се избират за релации препратките между блокове от програмата [Фиг. 4.]. Резултатният граф автоматично сортира възлите и съобразява разстоянието помежду им за да създаде удачна подредба на структурата.



Фиг. 4. Граф на логиката на изпълним код, постигнат с проекта

Резултатният граф и свързаната с него информация се съхранява в статична колекция от файлове, който може да бъде отворен с уеб браузър с възможност за изпълнение на JavaScript. Това дава възможност способностите на браузъра за хардуерно ускорение на графика да бъдат използвани при представянето на големи графи. Всеки възел може да бъде лесно преместен по преценка на потребителя. При нужда от допълнителна информация за блоковия възел, натискане на всеки граф води до таблица с известните свойства на блока, включително и деасемблиран код и локални променливи [Фиг. 5.].

function		
	parameterCount	0
	tags	

	decompiledFunction	undefined8 main(void)
		
		{
		  int iVar1;
		  long in_FS_OFFSET;
		  undefined local_58 [72];
		  long local_10;
		
		  local_10 = *(long *)(in_FS_OFFSET + 0x28);
		  puts("Howdy, so that\'s your first crackme challenge?");
		  printf("Then, tell me the password: ");
		  __isoc99_scanf(&DAT_00102054,local_58);
		  iVar1 = checkPassword(local_58);
		  if (iVar1 == 0) {
		  puts("Nope");
		  }
		  else {
		  puts("You are the chosen one!");
		  }
		  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
		  /* WARNING: Subroutine does not return */
		  __stack_chk_fail();
		  }
		  return 0;
		}

	localVariables	

dataType	length	minAddress	name	comment
undefined8	8	Stack[-0x10]	local_10	None
undefined1	1	Stack[-0x58]	local_58	None
undefined4	4	Stack[-0x5c]	local_5c	None
undefined8	8	Stack[-0x68]	local_68	None

Фиг. 5. Извадка от информацията за деасемблиран код с помощта на проекта

3. Използвани технологии

Основният програмен език, избран за разработката на проекта, е Python, благодарение на неговата гъвкавост и широката екосистема от налични

библиотеки. Използването на Python улеснява както процеса на разработка, така и интеграцията с външни инструменти, предоставящи приложно-програмни интерфейси (API). С помощта на тези интерфейси Python позволява лесно взаимодействие с разнообразни технологии, като същевременно предлага мощни възможности за автоматизация и обработка на данни, което значително ускорява реализацията и подобрява функционалността на проекта.

Ghidra е изключително гъвкав инструмент, когато става въпрос за работа с различни формати на изпълними и двоични файлове. При зареждане на файл, системата автоматично анализира неговата структура и идентифицира критични параметри, като архитектура на процесора (x86, ARM, MIPS и др.), формата на стойности (little Endian / Big endian), използвани секции и атрибути на файла. Ghidra позволява и ръчно задаване на тези параметри, ако автоматичното разпознаване не е успешно, което е полезно при анализ на обфускирани файлове или нестандартни формати. Зареждането на двоичния файл е първата стъпка, която осигурява основа за последващите етапи на декомпилация и анализ.

Ghidra Python API представлява мощен програмен интерфейс, който позволява автоматизация на Ghidra и използването му в безграфичен (headless) режим чрез скриптове, написани на Python 2.0. Тази функционалност значително улеснява интеграцията на Ghidra в по-големи аналитични потоци и автоматизирани системи за анализ на двоични файлове. За допълнително удобство и ефективност беше разработен интерфейсен модул, който автоматично извиква Ghidra в безграфичен режим с необходимите командни параметри. По този начин се елиминира необходимостта от ръчна работа през терминала на операционната система, като заявките за изпълнение се обработват директно и интуитивно чрез модула.

За създаването на графовидната репрезентация на кодовите блокове бяха използвани мощните Python библиотеки NetworkX и Pyvis, които осигуряват както формирането на графа, така и съхранението му в леснодостъпен и

организиран формат. Тези библиотеки позволяват ефективна работа с графови структури и визуализация на връзките между отделните кодови блокове.

За съхранението на извлечените данни бяха приложени различни технологии, всяка от които е избрана заради своите предимства в контекста на анализа. Деасемблирането на кода се извърши с помощта на C, който, благодарение на ниското си ниво, позволява преобразуването на асемблерни инструкции в основни команди, които са по-разбираеми за човека. Макар и близък до хардуера, C предлага значително по-високо ниво на абстракция в сравнение с Assembly и други архитектурно-зависими кодировки, което го прави идеален за този процес.

Известната информация за анализираната програма се съхранява във формат JSON, където всеки асемблерен блок е представен като обект. Тези обекти са обогатени с C-подобен код, както и с допълнителни данни за функции, локални променливи и друга контекстуална информация. Финалните графовидни репрезентации и блокови обекти се записват в статичен HTML формат, който е преносим, универсален и лесен за визуализация в различни среди. Тази структура осигурява както удобство при анализ, така и възможност за споделяне и по-нататъшна обработка.

4. Потенциални приложения

Изграждането на диаграми на кода и графи като карта на знанието е мощен инструмент за тестване и отстраняване на грешки в софтуера. Тези визуализации представят потока на управление и зависимостите между функциите, което позволява на разработчиците бързо да идентифицират проблемни области. Деасемблираният код допълва този процес, като осигурява достъп до ниско ниво на детайли, необходими за разкриването на сложни грешки, като неправилно използване на паметта или неочаквани зависимости. Чрез графовидни структури специалистите могат да симулират различни

сценарии на изпълнение на кода и да локализират бъгове, които биха останали незабелязани при стандартно тестване.

Графовете и диаграмите създават структурирана визуализация на логиката и архитектурата на кода, което значително улеснява документирането на софтуерните проекти. Те могат да служат като жива документация, която представя връзките между модулите, последователността на изпълнение и важните зависимости. Дисасемблираният код може да бъде добавен като детайлен източник на информация за специфични реализации, което е особено полезно при разработка на големи системи или при работа с наследен код. Това допринася за по-добра комуникация между екипите, както и за по-бързо въвеждане на нови членове в проекта.

Визуализациите и анализът на деасемблиран код са основополагащи за идентифицирането на уязвимости и проверката на сигурността на софтуера. Графовете могат да покажат потенциално рискови пътища в потока на управление, като например необработени изключения или незащитени точки за достъп. Дисасемблираният код разкрива скрити уязвимости, като неправилно управление на паметта или потенциални входни точки за атаки. С помощта на тези инструменти специалистите по киберсигурност могат да анализират и подсилват безопасността на програмите, като прилагат навременни корекции.

Изграждането на графи и диаграми е незаменимо при анализа на зловреден софтуер, тъй като те показват как зловредният код се разпространява и изпълнява. Диаграмите позволяват бързо идентифициране на критични функции, като тези, свързани със събирането на данни, комуникацията със сървъри или манипулирането на файлове. Дисасемблираният код разкрива алгоритмите, използвани от малуера, което е ключово за разбирането на неговото поведение. Тези анализи са от съществено значение за разработката на ефективни методи за защита и премахване на зловреден софтуер.

Графовидните визуализации и дисасемблираният код имат важно място в обучението на специалисти по програмиране и киберсигурност. Те осигуряват

практически подход за разучаване на архитектурни модели, алгоритми и работа на ниско ниво с компютърни системи. Чрез разглеждането на графи студентите могат лесно да проследяват взаимодействията в кода, докато дисасемблираният код дава детайлно разбиране за процеса на преобразуване от език с високо ниво на абстракция език в машинен код. Тези инструменти предоставят безценни знания за разработването на ефективни и безопасни софтуерни решения.

5. Следващи стъпки

В настоящия проект не е напълно имплементирана, но е изследвана и тествана базова функционалност за разширение чрез използване на големи езикови модели (LLM), които са специализирани в обработка на естествен език, трансформация и адаптация на съдържание, както и в оценка на структурирани данни. Основната цел на тези модели е да осигурят по-високо ниво на абстракция при анализа на изпълними файлове, което не само ускорява процеса, но и прави резултатите по-достъпни и лесни за интерпретация, дори за неспециалисти. Чрез интеграцията на LLM, анализът придобива ново измерение, което комбинира мощта на автоматизацията с разбираемостта и контекстуалността, необходими за ефективно вземане на решения.

Диаграмите от високо ниво и графите като карта на знанието (knowledge map) са мощни инструменти за откриване на уязвимости и злонамерен код. Те предоставят структурирана визуализация на връзките между различни модули, потока на управление и използваните ресурси, което позволява лесно идентифициране на необичайни или подозрителни поведения. Машинното обучение с езикови модели (LLM) допълва този процес, като автоматично анализира кода и изтъква потенциални уязвимости, като SQL инжекции, буферни препълвания или неподходящо управление на права за достъп. Комбинацията от визуализации и LLM позволява на специалистите да локализират и адресират проблемите по-бързо и по-точно.

С помощта на диаграмите и knowledge map графите аномалиите в кода могат да бъдат представени в ясен и разбираем формат. Визуалното представяне на тези аномалии помага на екипите да оценят тяхната тежест и потенциалния им ефект върху сигурността на системата. LLM инструментите допринасят, като автоматично генерират подробни отчети за намерените аномалии, включително обяснение за тяхната същност, местоположение в кода и препоръчителни действия за отстраняване. Тази автоматизация спестява време и намалява вероятността от човешки грешки при документиране на проблемите.

LLM може да бъде интегриран като помощник за подсказки, свързани със сигурността на кода. Докато разработчиците пишат или анализират код, езиковият модел може да предлага превантивни мерки за подобряване на сигурността, като използване на безопасни библиотеки, защита на чувствителна информация или предотвратяване на рискови операции. Чрез анализ на диаграмите и графовите структури, LLM може да идентифицира потенциално рискови точки и да предложи оптимални практики за тяхното управление.

Диаграмите от високо ниво, съчетани с LLM, са изключително полезни за рефакториране на кода. Въз основа на анализа на структурата на програмата, езиковият модел може да предложи промени за подобряване на четимостта, намаляване на сложността или премахване на излишния код. Например, LLM може да идентифицира дублиращи се функции или лошо структурирани зависимости между модулите и да препоръча оптимални начини за тяхното преструктуриране. Тези подсказки улесняват поддържането на кода и подобряват неговата производителност.

LLM инструментите предлагат уникалната възможност автоматично да създават описания на логиката и функциите на даден код. Чрез анализ на графите и кода, езиковите модели могат да генерират обяснения за това какво прави дадена функция, какви са очакваните входни и изходни данни и какъв е контекстът на изпълнение. Това е особено полезно за нови членове на екипа или

за документиране на сложни проекти, където разбирането на кода изисква значителни усилия.

Чрез анализа на контекста и структурата на кода, LLM може автоматично да идентифицира типовете данни и да предлага подходящи имена за променливите, съответстващи на тяхната функция [Фиг. 6-7.]. Например, вместо използването на общи имена като var1 или temp, моделът може да предложи по-описателни названия като user_age или transaction_amount, които ясно отразяват предназначението им. Това улеснява четимостта на кода и подобрява разбирането му както за текущите разработчици, така и за бъдещите поддръжници на проекта.

localVariables

dataType	length	minAddress	name	comment
undefined8	8	Stack[-0x10]	local_10	None
undefined1	1	Stack[-0x58]	local_58	None
undefined4	4	Stack[-0x5c]	local_5c	None
undefined8	8	Stack[-0x68]	local_68	None

Фиг. 6. Локални променливи на дизасемблиране код

Suggested Variable Name Changes with Explanations

1. `in_FS_OFFSET` → `fs_offset`
   This variable seems to represent the base of the frame pointer (stack frame), so `fs_offset` is a more descriptive name.
2. `local_58` → `user_input`
   This variable holds the input password, so `user_input` is a much clearer name.
3. `local_10` → `stack_canary`
   Since this variable is used to check the stack canary (a security mechanism), `stack_canary` is a more meaningful name.
4. `iVar1` → `password_check_result`
   This variable stores the result of the `checkPassword()` function, so `password_check_result` would make it clearer.
5. `DAT_00102054` → `password_format_string`
   This variable is the format string for `scanf`, so `password_format_string` describes its purpose better.

Фиг. 7. Препоръки от GPT-4o за преименуване на променливите

Сред тестваните модели в проекта бяха включени някои от най-известните и напреднали езикови модели за обработка на код, като GPT-2 от OpenAI и CodeBERT от Microsoft [5-9]. Целта на тяхното използване беше да обогатят изходния код, написан на C-подобни езици, и да предоставят пояснения за смисъла и контекста на различните части на кода. Тези модели имат способността да анализират сложни структури на код и да генерират текстови обяснения, които правят разбирането на програмата по-достъпно, особено за потребители с малък опит в програмирането. Освен тях, беше изпробван и програмно-приложният интерфейс (API) за директна връзка с по-нови и по-мощни модели като GPT-3.5-turbo и GPT-4, които бяха използвани за анализ на употребата на локални променливи в дисасемблираните функции. Тези модели предложиха описателни имена за променливите, които бяха съобразени със тяхната функция и начин на използване в контекста на изпълнимия код.

Освен че автоматично именуват променливите, тези модели също така генерират контекстуални обяснения, които улесняват разбирането на логиката зад кода и му придават по-висока четимост. Допълнително, в рамките на проекта, бяха проведени тестове с рефинирани версии на CodeBERT, специално настроени за задачи по ревюиране на код. Тези тестове включваха предложение за корекции на кода, като се взимат предвид най-добрите практики за програмиране, и оценка на сигурността на кода от гледна точка на киберсигурността. Използвайки тези модели, беше възможно да се открият потенциални уязвимости и слаби места в кода, които могат да бъдат използвани за атаки, като буферни препълвания или незащитени входни точки. Това допринася за значително подобряване на безопасността и устойчивостта на анализирания софтуер. Тези изследвания демонстрират как съвременните езикови модели могат да се използват не само за автоматизация на анализа на

код, но и за значително повишаване на сигурността и качеството на софтуерните решения.

6. Заключение

Настоящата курсова работа обобщава цялостния процес на анализ на изпълними файлове чрез декомпилация, графови репрезентации и интеграция на изкуствени интелектуални технологии. Разгледаните методи за декомпилация и деасемблиране позволяват дълбок анализ на ниското ниво на кодовите структури, осигурявайки по-добро разбиране на вътрешната логика на програмите. Чрез използването на инструменти като Ghidra, бе реализиран процес на декомпилация, създаване на графове и извличане на важна информация за програмните блокове, която е критична за разкриване на функциите, променливите и зависимостите в изпълнимите файлове.

В допълнение, визуализацията на кода чрез графове като карта на знанието добавя ново измерение към анализа на софтуера, като предоставя ясен и структуриран начин за разбиране на потока на изпълнение и зависимостите между различни части на програмата. Графовите структури не само че улесняват идентифицирането на проблеми и аномалии, но също така служат като основа за по-нататъшно разширяване и анализ.

Не по-малко важен е приносът на изкуствения интелект в тази работа. Използването на големи езикови модели, които имат значителен потенциал за принос в сферата на киберсигурността. Чрез интеграцията на тези модели с графовите визуализации, бе постигнато значително ускоряване на анализа и повишаване на разбирането на кода, което го прави по-достъпен и полезен не само за разработчици, но и за специалисти по киберсигурност.

7. Литература

1. Graph view - Obsidian Help. (n.d.). https://help.obsidian.md/Plugins/Graph+view

2. Alanazi, R., Gharibi, G., and Lee, Y. Facilitating program comprehension with callgraph multilevel hierarchical abstractions. Journal of Systems and Software 176(2021), 110945, https://doi.org/10.1016/j.jss.2021.110945
3. Symbolk. (n.d.). GitHub - Symbolk/Code2Graph: Towards converting multilingual source code into one language-agnostic graph representation. GitHub. https://github.com/Symbolk/Code2Graph
4. Binaryai. (n.d.). GitHub - binaryai/sdk: Get results of binaryai.cn using our SDK. GitHub. https://github.com/binaryai/sdk
5. openai-community/gpt2 · Hugging Face. (n.d.). https://huggingface.co/openai-community/gpt2
6. microsoft/codebert-base · Hugging Face. (n.d.). https://huggingface.co/microsoft/codebert-base
7. MRM8488. (n.d.). CodeBERT-base fine-tuned for insecure code detection. Hugging Face. https://huggingface.co/mrm8488/codebert-base-finetuned-detect-insecure-code
8. MRM8488. (n.d.). CodeBERT-base fine-tuned for code refinement. Hugging Face. https://huggingface.co/mrm8488/codebert2codebert-finetuned-code-refinement-small
9. Jiekeshi. (n.d.). CodeBERT-25MB for vulnerability prediction. Hugging Face. https://huggingface.co/jiekeshi/CodeBERT-25MB-Vulnerability-Prediction
10. Ghidra.python. (n.d.). https://ghidra.re/ghidra_docs/api/ghidra/python/package-summary.html
11. Headless Analyzer README. (n.d.). https://static.grumpycoder.net/pixel/support/analyzeHeadlessREADME.html
12. Kirillwolkow. (n.d.). Easy-CrackMe-Binary/crackme.c at main · kirillwolkow/Easy-CrackMe-Binary. GitHub. https://github.com/kirillwolkow/Easy-CrackMe-Binary/blob/main/crackme.c