Technologic Papers

דגשים לתכנות מאובטח

מאת אדיר אברהם

הקדמה

האם חשבת על הקוד שלך כ"קוד בטוח"? האם אתה מתכנת קוד מאובטח? מה זה בכלל ומה הבעיה, בעצם? כתיבת קוד לכשעצמו, בדר"כ מונחית מטרה מסויימת - התוכנית צריכה לבצע את מה שביקשו מאיתנו שהיא תעשה. אך האם זה מספיק? לכאורה, התוכנית שלנו מבצעת את מה שביקשו מאיתנו, אך האם חשבנו באמת על כל האפשרויות? מה יקרה אם נאפשר הכנסת קלט גדול מדי, או סתם נאפשר גישה לספריה בתוך המערכת כי נוח לנו? ומה אכפת לנו בעצם?

במאמר זה אנסה להסביר את הבעיות בכתיבת קוד "פשוט", ללא התחשבות באבטחתו, וכן אסביר כיצד לכתוב קוד בטוח. במאמר זה אתייחס בעיקר לעקרונות בשפת C, אך העקרונות שמוצגים כאן, מתאימים כולם או חלקם גם לשפות נוספות, וחשוב להתייחס אליהם בזמן כתיבת הקוד. נעבור על דברים שצריך לעשות (ואנחנו, כנראה, לא תמיד עושים אותם או חושבים עליהם) ולא פחות חשוב - למה צריך לעשות אותם.

Validation של הקוד

שימוש נכון בשורת הפקודה:

הרבה תוכניות מקבלות קלט משורת הפקודה. פונקציות כגון getuid ו-setgid עלולות להיות בשימוש ע"י משתמש לא אמין. לכן, פקודות אלו צריכות להגן על עצמן משימוש לא נאות דרך שורת הפקודה. תוקפים עלולים לשלוח נתונים דרך שורת הפקודה, ולכן יש צורך בוידוא הקלטים ולא לתת שימוש ישיר בפונקציות עצמן, ללא וידוא קלט נכון והגיוני שמגיע ישירות מהמשתמש.

משתני סביבה:

בתור ברירת המחדל, משתני הסביבה מתקבלים בירושה מתהליך האב. למרות זאת, כשתוכנית מריצה תוכנית אחרת, היא יכולה לשנות את משתני הסביבה המקוריים למשתני סביבה משלה, וע"י כך ליצור שליטה בסביבת העבודה. נרצה למנוע את אופציית השינוי.

שמות קבצים:

לא מעט פעמים לא שמים לב לשמות קבצים, למרות שיש להם משמעות רבה הן כקלט והן כסטנדרט. נרצה להמנע (בהעדר סיבה מוצדקת) מקליטת שמות הקבצים המכילים את ".." (ספריה אחת למעלה), וכן נרצה להמנע משינוי או יצירת ספריה חדשה ע"י איסור השימוש בתו "/". נרצה גם להמנע משמות קבצים המכילים wildcards כגון "*", "?" וכדומה. מניעות נוספות הן שימוש בקו מוביל ("-") אשר עלול להתפרש כפרמטר, שימוש ברווחים אשר עלול להתפרש כשני קבצים נפרדים, שימוש בתוים שאינם תואמים לתקן UTF-8 ועוד.

תכני הקובץ:

מה יכיל קובץ? נניח שנרצה לקלוט נתונים מתוך קובץ אשר ינתן ע"י המשתמש. מה יהיה בו? אך ורק את מה שנגדיר כמשהו חוקי בלבד, כדי למנוע את הבעיות מסעיפים א-ג וכן בעיות נוספות (באגים). ואם ניתן למשתמש קובץ בעצמנו? נרצה כמובן שהוא לא ישנה את הקובץ (שמו, עריכה/מחיקה שלו וכו'), ולכן נרצה למנוע זאת ממנו ע"י הגבלות מתאימות.

מניעת Buffer Overflow

רבות נאמר ונכתב על Buffer Overflow, ובכל זאת, חשוב להזכיר את הנושא. נדגיש את הנושאים העיקריים בשפת C ו- C++. Buffer Overflow נגרם כאשר קלט באורך מסויים נכתב לתוך buffer שגודלו קבוע וקטן יותר מאורך הקלט. קלט כזה יכול להתקבל ע"י המשתמש ישירות, וכן מגורמים נוספים אשר נובעים מעיבוד הקובץ. אם ה-buffer מכיל משתני סביבה של כותב התוכנית, נוכל לדרוס אותם ע"י buffer overflow וכן כתיבת משתני סביבה משלנו. התוצאה היא שנוכל להריץ קוד זדוני כלשהו. שפות תכנות גבוהות יותר חסינות מכך, מפני שהן מכילות מנגנוני הגנה שונים כגון שינוי גודל ה- buffer או הכנסה לתור אחר, כשישנה חריגה מהגודל המקורי. לשפת C אין הגנה כזו, וניתן ליצור בעיות דומות ע"י שימוש לא נכון בשפת C++.

מתכנתי שפת C צריכים להמנע משימוש בפונקציות אשר לא בודקות חסם עליון של buffer (אשר אותו ניתן לעבור וע"י כך לעשות buffer overflow). פונקציות לדוגמא הן vsprintf(), sprint(), strcat() וכן gets(). אלו צריכות להיות מוחלפות עם snprintf(), strncat(), strncpy() ו- fgets() בהתאמה. גם משפחת scanf() מסוכנת וכן הפקודות strecpy(), streadd(), getpass(), getopts(), realpath() ו-strtrns() שמקבלות פרמטרים ללא חסם על ה- buffer. בנוסף לחסם על גודל ה- buffer, נרצה לבדוק שהגודל לא שלילי, מכיוון שאם הנתונים הם signed, נוכל לגלוש מגודל הנתונים ע"י מתן אורך קלט גדול מדי.

לדוגמא:

```
char *buf;
int i, len;
read(fd, &len, sizeof(len));
if (len > 9000) { error("too large length"); return; }
/* len < 0 ... */
buf = malloc(len);
read(fd, buf, len); /* len is casted to unsign and overflows */
```

עקרונות תכנות נכונים לאבטחת תוכנות

כמהנדסי תוכנה, נרצה לשמור על מספר עקרונות בסיסיים כאשר אנו מתכננים ויוצרים תוכנה פוטנציאלית. חלק מהעקרונות מבוססים על תכנות נכון באופן כללי, אך כל המצויינים כאן חשובים גם לאבטחת הקוד הנכתב לשימוש נכון ולא זדוני.

מינימום הרשאות:

כל משתמש וכל תוכנית צריכה לפעול עם מינימום עם ההרשאות האפשרי. עקרון זה יקטין את הסכנה מפני טעות או תקיפה של משתמש אשר יוכל לקבל הרשאות לבצע דברים במערכת שהוא לא צריך לעשות. עקרון זה גם יקטין את מספר האינטרקציות עם תוכניות בעלות פריבילגיות גדולות יותר כך שפעולות לא מכוונות ולא נאותות לא יבוצעו ע"י תוכניות אלו. נרחיב את עקרון זה ונאמר שרק החלק הקטן ביותר של התוכנית שצריך לקבל פריבילגיות גדולות יותר, אכן יקבל אותן. בכל שלב אחר, הפריבילגיות תהיינה נמוכות.

בדיקת הרשאות:

כל ניסיון גישה חדש, אשר דורש פריבילגיה גבוהה יותר, חייב להבדק. בדיקה זו חייבת להיות מוגנת משינוי או תקיפה. למשל, בתכנון שרת חדש, נרצה שהשרת יוכל לבדוק את כל המשתמשים (clients) שרוצים לגשת למערכת, אך שהם לא יוכלו לשנות את הבדיקה הזו, להתחמק ממנה או חלילה לשנות את השרת או את שאר המשתמשים.

מנגנוני אל-כשל:

מה נעשה עם המערכת נכשלת? ומה נעשה אם הקלט הוא שגוי? נרצה במקרה כזה למנוע גישה. לא נשאיר את המערכת כמות שהיא. כאשר משתמש מסויים נותן למערכת קלט שגוי, יש למנוע ממנו את הגישה לנסיון נוסף. בהנחה שמדובר במשתמש תמים (נניח, כזה ששכח סיסמא), ניתן לו את האפשרות לנסות עוד מספר פעמים מועטות עד שנחסום לו את הגישה לחלוטין. במקרה כזה, נוכל לשלוח הודעת שגיאה מתאימה אשר תבהיר שהמשתמש נחסם, אך נשתדל לא לשלוח הודעה מפורטת מדי (כגון הודעת

Debugging) אשר תתן לתוקף פוטנציאלי אינפורמציה מפורטת מדי, כזו שתאפשר לו בסופו של דבר למצוא פרצות אחרות במערכת ולנצל אותן.

מינימום זמן לפריבילגיות גבוהות:

במידה ויש צורך לתת פריבילגיה גבוהה יותר בפרק זמן מסויים (למשל, עבור יצירת קובץ חדש), יש להגבילו בזמן כדי להבטיח שהפריבילגיה תתקבל אך ורק לזמן הדרוש לה ולא מעבר. לאחר השימוש בפריבילגיה, או לאחר אי-שימוש בפרק זמן מסויים, יש להגביל את הפריבילגיה שניתנה.

הגבלת המידע הנגיש:

במידת האפשר, יש להקטין למינימום את כמות המידע הלא-רלוונטי שניתן לגשת אליו. למשל, בעת גישה לקריאה ממסד נתונים, רצוי לאפשר את קריאת המידע הרלוונטי (שאותו המשתמש בסופו של דבר אמור לקרוא).

הגבלת כמות המשאבים המאופשרת:

יש למנוע ממשתמש לגזול את משאבי המערכת, ע"י שימוש מסיבי בהם. כלומר, במידה וישנו שימוש רב מדי במשאב מסויים, יש להוריד ממנו את ההרשאות המתאימות לכך, או להגביל את המשאב ע"י מסירתו למשתמש אחר, כדי למנוע רעב בתוך המערכת.

הקטנת הפונקציונליות של הרכיב:

יש להקטין את פונקציונליות הרכיב. בצורה כזו, נוכל לאפשר פונקציונליות ספציפית אשר תתן למשתמש אך ורק את מה שהוא צריך ולא מעבר. דבר זה מאפשר בסופו של דבר שליטה ובקרה במה שנעשה במערכת מצד אחד, ומניעת פריבילגיה גבוהה יותר מתי שלא צריך מצד שני.

מניעת Race Conditions:

גישת שני תהליכים למשאב משותף תוך פרק זמן קצר מאוד, עלולה לגרום למצב של Race Condition. במצב זה, תגובת המערכת עלולה להיות לא מוגדרת, מה שעלול בסופו של דבר לגרום לפרצה במערכת. על-מנת להמנע ממצב זה, יש למנוע מפעולות לא אטומיות אשר יאפשר לשני תהליכים לנסות להשתמש באותו משאב באותו פרק זמן.

מצב דומה עלול לקרות משני תהליכים שונים שכן אמורים להשתמש באותו פרק במשאב משותף (למשל, אחד כותב קובץ והשני קורא ממנו באותו פרק זמן). נרצה במקרה כזה לסנכרן ביניהם, כך שלא שניהם יגשו באותו פרק זמן.

להלן דוגמא להרצה (תוך שימוש בפונקציות לא בטוחות):

```
#include <stdio.h>
static void charatatime(char *);
int main(void)
{
	pid_t pid;
	if( (pid = fork()) < 0)
	{
		perror("Fork error");
		exit(1);
	}
	else if( pid == 0)
	{
		charatatime("Output from child\n");
	}
	else
	{
		charatatime("Output from parent\n");
	}
	exit(0);
}

static void
charatatime(char *str)
{
	char *ptr;
	int c;
	/* Send output to the buffer as soon as possible */
	setbuf(stdout,NULL);

	for(ptr = str; c = *ptr++; )
		putc(c, stdout);
}
```

שימוש בקבצים זמניים:

נרצה להשתמש בזהירות בקבצים זמניים. קבצים זמניים נשמרים בדר"כ בספרייה ייעודית. תוקפים יצרו שם קובץ בספריה הזמנית לקובץ שהם מצביעים אליו. לכן, אנו נרצה לבדוק האם שם קובץ כזה קיים (ואם כן, אז ניצור קובץ בשם אחר), ואם לא אז ניצור אותו וניתן לו את ההרשאות המתאימות (אחרת, התוקף יוכל לשנות את הקובץ להצביע אל הקובץ שלו). נרצה ליצור קבצים עם שמות "אקראיים" כדי שהסבירות לטעויות כאלה תקטן. פונקציה שיוצרת קבצים זמניים מתאימים היא mkstemp(). ניתן לה שם קובץ מהצורה "fileXXXXXX", והיא תחליף כל "X" עם המספר הפנוי, ובצורה כזו ימנע שימוש בשם קובץ זמני שכבר קיים, וע"י כך ימנע גם Race Condition בין בדיקת הקיום של הקובץ לבין פתיחת קובץ קיים ושימוש בו.

קריאה זהירה לספריות חיצוניות:

כיום, אין תוכנה שהיא מכילה אך ורק עצמה. ברוב מכריע של המקרים, אנו נשתמש בספריות אשר נוצרו בשבילנו - פונקציות שירות סטנדרטיות או מימושים מורכבים אשר בוצעו בהצלחה ע"י אחרים. פונקציות אלו נמצאות כחלק סטנדרטי ממערכת ההפעלה, ספריות תוכנה ועוד. נרצה להזהר בעת שימוש בהם, כדי למנוע את הבעיות שנכתבו בסעיפים הקודמים.

קריאה רק לספריות שידועות כ-"בטוחות":

לפעמים ישנו קונפליקט בין אבטחה לבין פיתוח סטנדרטי בעזרת אבסטרקציות ושימוש בקוד קיים. לעתים קוד קיים עשוי שלא להיות ממומש בצורה מאובטחת (למשל, ע"י שימוש בעקרונות הרשומים כאן) וזה לא יאמר במפורש. בצורה כזו אפילו אם מימשת קוד בצורה מאובטחת, עבודתך עלולה לרדת לטמיון. לשם כך, במידה והתוכנה שלך חייבת להיות בטוחה, יש צורך לממש את החלקים שנחשבים לא בטוחים, או לחילופין יש לבדוק את החלקים שעלולים להיות בעייתיים כדי להמנע מפירצה פוטנציאלית.

קריאה אך ורק עם פרמטרים מאומתים:

יש לאפשר קריאה לספריות חיצוניות, אך ורק לאחר שהפרמטרים אותם שולחים נבדקו ואומתו. כאן, יש לבדוק שהמידע שנשלח אכן המידע שאנו רוצים להשלח, ולא מידע שגוי, כדי למנוע חשיפה לטעות מצד הספריה החיצונית (וע"י כך ליצור, למשל, SQL Injection).

בדיקת כל החזרות מקריאות מערכת:

כל קריאת מערכת שיכולה להחזיר תנאי שגיאה, יש לבדוק את אותו תנאי השגיאה כדי לראות מה לא היה בסדר ולפעול בהתאם. במידה ולא נבדוק ונמשיך הלאה, עלול להווצר מצב בו משתמש מסויים קיבל גישה למשאב שהוא לא היה צריך לקבל.

פידבק ואינפורמציה מהמערכת: (מיעוט באינפורמציה מיותרת)

כפי שנכתב בהתחלה, יש צורך למעט באינפורמציה אשר מוחזרת מהמערכת. במידה ומדובר במשתמש לא אמין, יש צורך להחזיר הודעת שגיאה או הצלחה בלבד. במידה והמשתמש אמין, יש צורך להחזיר הודעה על מה היתה שגיאה, אך לא לפרט היכן היא היתה (למשל, "שורה מספר 132 גרמה לשגיאה מכיוון ששם הקובץ הכיל 18 תוים ולא 12").

טיפול בפלט:

במידה ולמשתמש יש אפשרות להחזיר פלט בכל רגע נתון, מצב זה עלול להאט את המערכת ולגרום ל-Denial of Service. לכן, יש צורך למנוע כל פרק זמן מסויים את כמות הפלט המוחזרת מהמערכת עבור משתמש ספציפי.

הגבלת הפורמט של נתוני הקלט:

יש להגביל קבלת קלט "חופשי", כזה אשר יכול ליצור פרצות לוגיות במערכת. במקום זה, יש ליצור פורמט מסויים אשר ידוע מראש למשתמש ובו הוא חייב לעמוד. בנוסף, יש להשתמש בפונקציות בטוחות לשם כאן, כדי למנוע מהתוקף לשנות את הפורמט וע"י כך להכניס איזו מחרוזת שירצה למערכת.

סיכום

תכנון ומימוש מערכת בטוחה היא משימה לא פשוטה, במיוחד בשפות כמו C ו-C++ בהן המערכות למניעת פרצות זדוניות מוגבל או לא קיים. במאמר זה, צוינו העקרונות לשמירה על קוד מאובטח. הקושי האמיתי, למעשה, הוא ביצירה של תוכנה אשר תדע להגיב נכון לכל סוגי הקלט האפשריים ומשתני הסביבה אשר משתמשים זדוניים ו"תמימים" מנסים להכניס אל המערכת. מפתחים שמשתמשים בעקרונות אלו, צריכים להבין את כל ההשלכות שנכתבו כאן, ולהעמיק בנושאים אלה בהתאם לפונקציות שהם כותבים או משתמשים בהם. ניתן לסכם את העקרונות בצורה הבאה:

- בדוק את כל הקלט שאתה מקבל, כולל אלו משורת הפקודה, משתני סביבה ונתונים נוספים.
- אל תסמן רק קלט "רע". דע גם לסמן מהו קלט "טוב".
- מנע Buffer Overflow בכל מקום. שים לב במיוחד לקלטים ארוכים ואל תתן להם את האפשרות להשתלט על פונקציונליות המערכת שלך.
- זכור לבנות את התוכנית שלך נכונה - מנע פריבילגיות גבוהות, אתחל את המערכת עם פרמטרים נכונים ובטוחים, תכנן מה תעשה כשישנו כשל של המערכת, מנע Race Conditions והשתמש בערוצים בטוחים בלבד.
- השתמש בזהירות בקריאות מערכת לספריות חיצוניות.
- החזר אינפורמציה מהמערכת בזהירות, רק את מה שצריך ולא מעבר. אל תחשוף נתונים פנימיים.