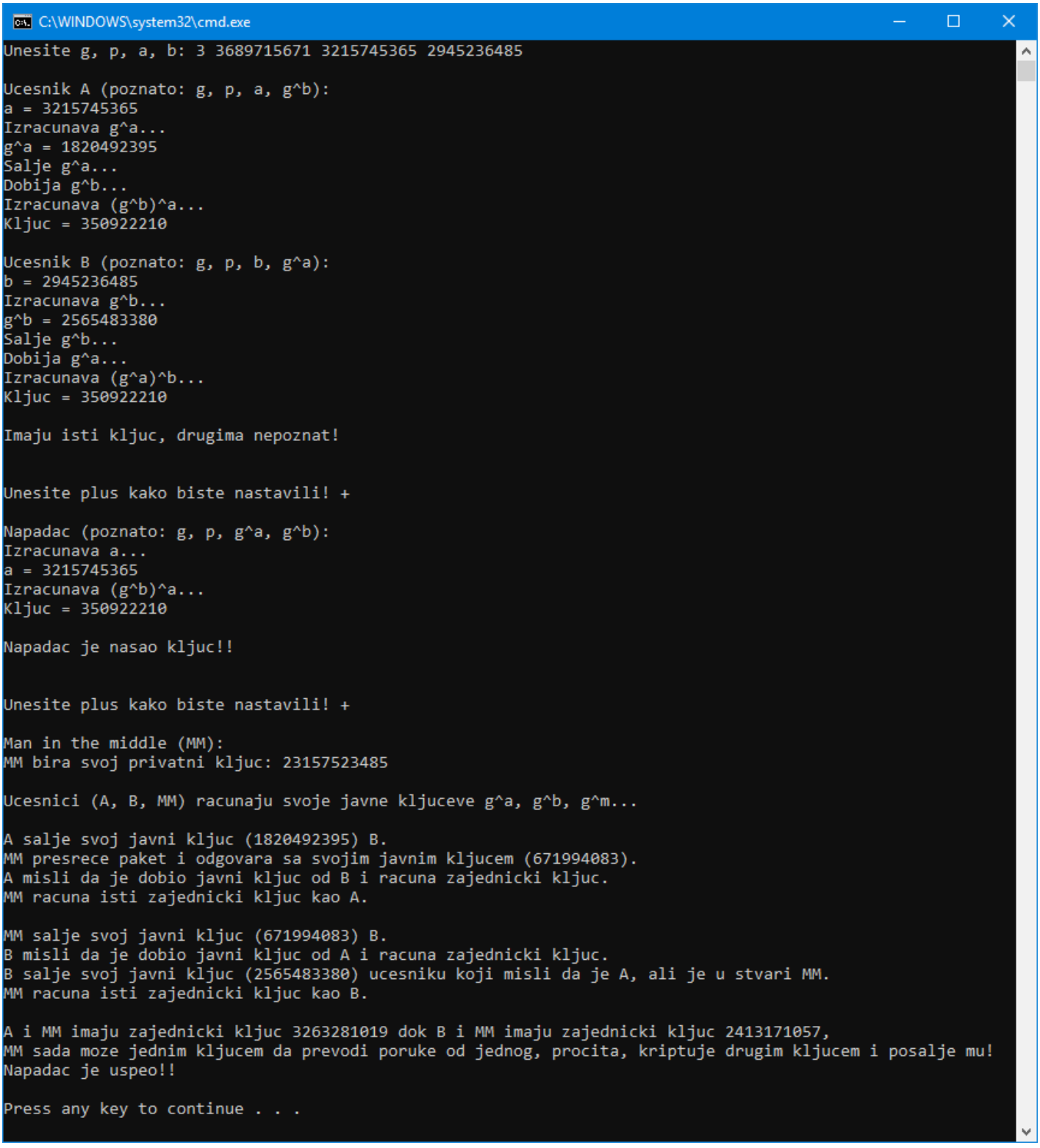
Diffie-Hellman key exchange

Student: Marko Mićović
2016/0633
Profesor: dr Predrag Ivaniš

Osnovi Telekomunikacija (OTR)

Beograd, 2019

Rezultat izvršavanja



Kratak Opis

Diffie-Hellman je sigurna metoda razmene ključeva koja ne zahteva upotrebu sigurnih kanala razmene. Učesnici javnim kanalima šalju svoje javne ključeve i samo oni (u realnoj količini vremena) mogu doći do zajedničkog ključa.

Metoda koristi osobinu nekih matematičkih funkcija koje su trivijalne za rešavanje, ali im inverzna funkcija zahteva daleko veću računarsku moć. U originalnoj implementaciji (kao i u ovom projektu) se koristi stepenovanje nad konačnim poljem.

Ovo je samo simulacija jedne takve razmene i nije pogodna za praktičnu upotrebu jer podržava maksimalno 32-bitne veličine konačnog polja. Za veće vrednosti dolazi do prekoračenja i dobijaju se netačne vrednosti. Ključevi tako male veličine se mogu izuzetno brzo razbiti čak i sa današnjim računarima potrošačkog razreda.

Sama simulacija objašnjava šta se dešava za vreme jedne takve razmene kao i za vreme *Man in the Middle* napada. Ovaj izveštaj će objasniti matematičku pozadinu simulacije.

Sve funkcija su u konačnom polju i date su bez komentara prisutnih u izvornom kodu.

Oprez - Napadač možda ne nadje isti privatni ključ koji je učesnik A izabrao, ali on svakako dobija isti zajednički ključ što se pokazuje sledećom jednakošću:

$(g^b)^{a1} = g^{a1b} = (g^{a1})^b = (g^a)^b = g^{ab}$

(*a1* – ključ koji napadač pronalazi kada traži ključ *a*)

Izvorni Kod

- Tipovi

uint64 – Skraćena predstava tipa *unsigned long long*

- Stepenovanje

```
uint64 powmod(uint64 g, uint64 a, uint64 p) {
	uint64 res = 1;
	while (a) {
		if (a % 2) {
			//a is odd
			res = (res*g) % p;
		}
		//a is even
		a = a >> 1;
		g = (g*g) % p;
	}
	return res;
}
```

Rezultat funkcije jednak je [$g^a \ mod \ p$].

Koriste se osobine stepenovanja [$g^a = g*g^{a-1}$] i [$g^{2b} = (g^2)^b$].

Prvom osobinom neparni stepen svodimo na parni kako bismo mogli da primenimo drugu osobinu i time skratimo vreme izvršavanja funkcije.

Deljenje stepena na dva se vrši pomeranjem u desno za jedan bit jer je bitsko pomeranje daleko brže od operacije deljenja.

Ovaj proces ponavljamo sve dok stepen ne svedemo na 0.

- Provera prostog broja

```
bool isPrime(uint64 p) {

	if (p <= 3) return p > 0;
	if (p % 2 == 0 || p % 3 == 0) return false;

	for (uint64 i = 6; i <= sqrt(p); i+=6)
		if (p % (i + 1) == 0 || p % (i - 1) == 0) return false;
	return true;
}
```

Svi brojevi veći od *sqrt(p)* koji dele *p* imaju faktor manji od *sqrt(p)*. Zbog te osobine ne moramo da proveravamo da li je broj deljiv sa brojevima većim od *sqrt(p)*.

Svi brojevi koji nisu deljivi sa 2 ili 3 su oblika *6*k+1* ili *6*k-1*. Zbog toga proveravamo deljivost samo sa brojevima tog oblika.

- Logaritmovanje

```
uint64 logmod(uint64 g, uint64 a, uint64 p) {

	uint64 res = 0;
	uint64 gpowres = 1;
	while (res < p) {
		if (gpowres == a) return res;
		res++;
		gpowres = (gpowres*g) % p;
	}
	return -1;
}
```

Rezultat funkcije jednak je [$log_g a \ mod \ p$].

S obzirom da se radi o diskretnom logaritmu znamo da rezultat mora biti pozitivan broj manji od p. Prostim ispitivanjem svih mogućih vrednosti dolazimo do rešenja.

Ova funkcije se u programu ne koristi. Data je jer pokazuje jednostavan način računanja logiritma nad konačnim polje takozvanom *naive brute force* metodom.

- Baby-step Giant-step

```
uint64 logmod_babygiant(uint64 g, uint64 a, uint64 p) {

	uint64 n = (uint64)sqrt(p) + 1;
	uint64 gn = powmod(g, n, p); //g^n

	unordered_map<uint64, uint64> values;
	uint64 cur = gn; // cur = g^(n*i)
	for (uint64 i = 1; i<n; i++) {
		if(!values[cur])
			values[cur] = i;
		cur = (cur*gn) % p;
	}

	cur = a; // cur = g^j * a
	for (uint64 j = 0; j<n; j++) {
		if (values[cur]) {
			return values[cur]*n - j;
		}
		cur = (cur*g) % p;
	}
	return -1;
}
```

Rezultat funkcije jednak je [$log_g a \ mod \ p$].

Problem pronalaženja pozitivnog broja x manjeg od p delimo na problem pronalaženja dva pozitivna broja i i j koji su manji od $n = sqrt(p)$.

$x = i*n - j; \quad g^x = a; \quad g^{i*n-j} = a; \quad g^{i*n} = g^j*a$

Izračunavamo sve moguće vrednosti g^{i*n} i čuvamo ih u mapi koja slika g^{i*n} u i. Za svaku moguću vrednost j proveravamo da li je g^j*a jednako nekoj od vrednosti iz mape. Kada nadjemo poklapanje izračunavamo x kao [$x = i*n - j$].

Ovaj algoritam je niže kompleksnosti od odnosu na *naive brute force* algoritam tako da u simulaciji primenjujemo njega.